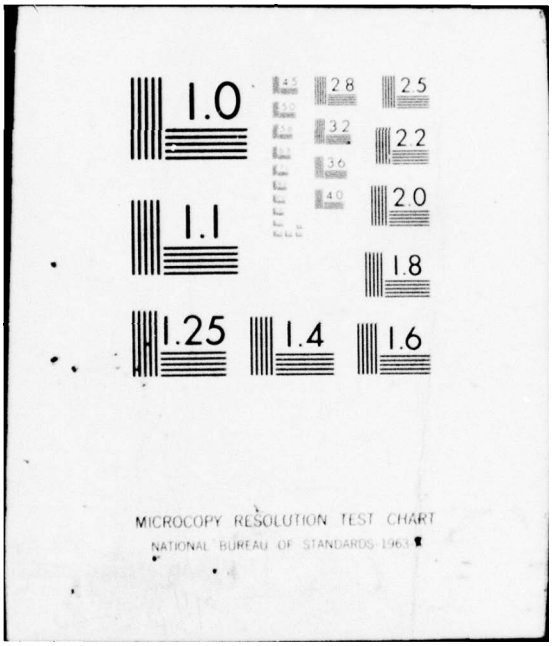
AD-A032 803 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OHIO SCH--ETC F/G 9/2
EFFICIENT MULTIPLE PROCESSOR COORDINATION.(U)
NOV 76 B E BAKER

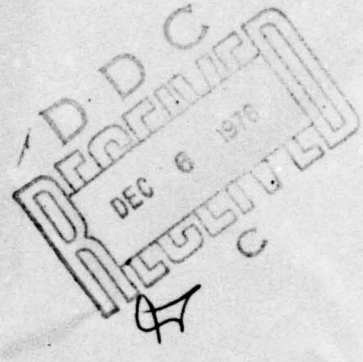
UNCLASSIFIED DS/EE/76-1 NL

1 OF 3
AD A032803


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963

DS/EE/76-1

(1)

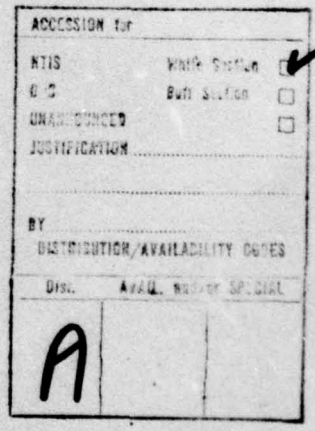


EFFICIENT MULTIPLE PROCESSOR COORDINATION

DISSERTATION

DS/EE/76-1

Bob E. Baker
Captain USAF



Approved for public release; distribution unlimited

DS/EE/76-1

EFFICIENT MULTIPLE PROCESSOR COORDINATION

DISSERTATION

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

by
Bob E. Baker, B.S., M.S.
Captain USAF

November 1976

DS/EE/76-1

EFFICIENT MULTIPLE PROCESSOR COORDINATION

by

Bob E. Baker, B.S., M.S.

Captain USAF

Approved:

Gary B Lamont 16 Nov 1976
Chairman

James B. Peterson 16 Nov 1976

Henry E. Orr 16 Nov 1976

Thomas E. Reeves 16 Nov 1976

Accepted:

J S Przemieniecki 16 Nov 1976
Dean, School of Engineering

Preface

I wish to thank my advisor, Professor Gary B. Lamont, for his able guidance and my readers, Professors George Orr, J. B. Peterson, and Edward Reeves, for their helpful suggestions. I am especially grateful to Professor Robert D. Dixon of Wright State University for carefully studying my proof of correctness of the mastermode/normalmode algorithm and for discovering a flaw in an earlier version of the algorithm. Finally I wish to thank my wife Tommie for her encouragement and patience during the course of this research.

The work reported herein was sponsored by the Air Force Avionics Laboratory (AFAL) and made extensive use of the DECsystem-10 computer facility at AFAL. This dissertation was set and typed by the RUNOFF utility program on AFAL's DECsystem-10.

Contents

List of Figures

Abstract

In recent years the concept of level-structured system organization has received growing acceptance as a computer operating system design technique. This approach treats the hardware of a computing system as the bottom level of a multilevel system which will be built up one level at a time. Each level creates an abstraction of the next lower level by implementing a new "virtual machine" which provides some feature not previously available.

This dissertation presents efficient solutions to certain problems of processor interaction which must be faced by the system designer at the lower levels of a strictly level-structured operating system. A loop-free algorithm is developed which solves Dijkstra's "critical section" problem, based on a rudimentary processor delay mechanism and the instruction set of a typical large computer of conventional architecture (DECsystem-10). A completely rigorous proof of the algorithm's correctness is given. The algorithm is used as the basis of a systematic procedure for obtaining efficient solutions to general processor coordination problems. The procedure is illustrated by solving several of the well-known reader/writer problems and a more complex new problem, the producer/consumer/distributor problem, which involves simultaneous use of several independent shared resources.

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND.

In recent years, remarkable advances have been made in the construction of large digital computing systems. Such systems are presently available whose capabilities far exceed those of the computing systems in use a decade ago. Current large-scale systems typically provide such advanced features as parallel interactive service of numerous independent users, virtual memory techniques which allow a user's memory address space to exceed the physical memory capacity of the system, increased computational power through the use of two or more central processors which operate concurrently, advanced on-line data base management capabilities, and sophisticated resource management facilities which allow the most efficient use of various peripheral devices connected to the system (De74, Pr75). Such features have proved to be extremely useful and have come to be accepted and even demanded by the users of digital computing systems. There seems little doubt that these and other advanced features will be incorporated into future computing systems.

The implementation of advanced features such as those described above has generally been accomplished by increasing the complexity of the control programs known collectively as the computer's "operating system" or "executive" or "monitor". This added complexity has led to the appearance of a number of serious problems which are presently among the foremost concerns of operating system designers. Large operating systems have become enormously expensive to build, with development times which are long and unpredictable. The system never becomes a finished product, because programming "bugs" appear continually and must be eliminated. Modifications to enhance system performance are difficult to install, and often have unforseen effects which appear much later as new bugs. Simply documenting the changes made in the software by the operating system developer and distributing that documentation to all users can present considerable difficulties.

Computer programmers have always faced problems in trying to establish and maintain the correctness of their programs, but these problems seem to have taken on an added dimension in the development of large and complex computer operating systems. This is due largely to the occurrence in such systems of a phenomenon which programmers have not often had to treat in depth previously: the simultaneous execution of many interacting programs. Such situations arise in various ways in the design of operating systems,

ranging from the apparent concurrency obtained by multiprogramming a single central processing unit (a technique which must by now be considered commonplace) to the real concurrency resulting from the presence of more than one central processor (Lo72). A predominant characteristic of a typical modern operating system for a large digital computer is that in order to understand it, one must be prepared to keep track of many activities which go on at the same time and which interact with one another. Thus the designers of a large operating system, who must presumably be the first to understand it, are faced with a monumental task if they wish to develop a system in which all possible interactions have been taken into account. Judging from experience with actual operating systems, this goal must be nearly unattainable.

1.2 LEVEL-STRUCTURED SYSTEM ORGANIZATION.

A significant step toward overcoming the problems created by increasing complexity and concurrency was taken by E. W. Dijkstra when he introduced in 1968 the idea of a level-structured, or hierarchical, system organization (Di68a,b). The operating system designer who wishes to adopt this approach views the physical devices (i.e., the hardware) of his computing system as the lowest level of a multilevel system which is to be built in stages from the bottom up, one level at a time. The structure which results

when a new level has been added is regarded as an "abstract" or "virtual" machine whose characteristics differ in some respect from those of the system which existed prior to the addition of that level. Once the designer has built a new level and tested the operation of the resulting virtual machine, he can proceed as if that virtual machine were the real machine for which he is designing an operating system. That is, he can forget about all lower levels and proceed with the development of a new level as though he were starting over with a real machine having the characteristics of his current virtual machine. Of course properly carrying out the testing mentioned above, although not our immediate concern, is by no means a trivial problem.

If the system designer is to realize a positive benefit from the hierarchical approach just described, he must see to it that the virtual machines created at successively higher levels are not merely different but are in some respect better machines than those which have been "left behind" at lower levels. Thus in deciding how to build the next level, the designer identifies some capability which he would like his system to have but which does not exist in his current virtual machine (for example, the ability to handle data files in a logical rather than a physical manner). He then implements that capability in software or firmware, creating an abstraction of his current virtual machine which becomes a new virtual machine at the next

higher level. This is the process referred to earlier as "building a new level."

The level-structured organization just described allows a system designer to create a sequence of virtual machines with progressively more desirable operating characteristics. At the same time it greatly reduces the amount of detailed system structure which the designer must be able to manage intellectually at a given time. Thus the designer of a given level is only concerned with providing a suitable virtual machine to the next level in terms of the virtual machine provided him by lower levels. He must know the characteristics of the latter machine, but does not have to be concerned with how it is implemented. Provided the increase in abstraction (i.e., the complexity of the added features) between adjacent levels is not too great, it should be possible for the designer to become thoroughly convinced of the correctness of the level he is designing before proceeding to a higher level, thus assuring himself of always having a properly functioning virtual machine with which to work.

A number of operating systems have been designed in which the concepts of level-structured organization have been applied more or less strictly (Di68a, Br70, Li72, Ak72, Se72). The development of specialized hardware mechanisms to support a level-structured operating system has also been proposed (Be75). In the present investigation we will

concentrate on finding efficient solutions to certain critical problems which must be faced at the lower levels of a strictly level-structured organization by a system designer who is constrained to work with a specified computing system whose hardware does not include explicit level-structuring mechanisms.

1.3 A POSSIBLE SECOND LEVEL.

In the preceding section we described how a level-structured system organization could help to solve the problems faced by an operating system designer due to the rapid increase in complexity as additional capabilities are added to the system. We have not, however, considered the question of how a level-structured design influences the problems which result from the presence of concurrent activity in the system. We do not intend to let this question escape our scrutiny. Indeed, the remainder of this dissertation may be considered an elaboration upon this particular issue.

As a first step toward understanding the relationship between level-structured system organization and concurrent activity, we will describe a virtual machine which might appear at the second level of a level-structured system design based on a specified collection of physical devices. Of course we do not have to worry about designing the first level; the equipment manufacturer did that when he built

the hardware. That is, we regard the physical machine as the first level of our contemplated level-structured system. We said that the virtual machine to be described is one which "might appear" at the second level because the level-structured design approach allows the system designer complete freedom in selecting the abstraction he wishes to make in proceeding from one level to the next. Hence many different system designs could result from the same first level, depending on the choices made by the system designer as he implements successive levels. The following description of a possible second level will hopefully serve to make the level-structured design concept more concrete for the reader. It is also intended to introduce the environment in which we will carry out the rest of this investigation.

For the physical machine upon which higher levels will be built, we will consider a typical large-scale computing system containing a single central processing unit (CPU) of conventional architecture. More specifically, we will base our design example upon a bottom level consisting of a Digital Equipment Corporation DECsystem-10 computer containing a Model KI10 CPU. The choice of this botton level is a result of the availability of such a system to the author for experimental work. The instruction set of the KI10 CPU includes arithmetic, logical, program control, and register and memory transfer operations typical of such processors. It also includes a number of read-modify-write

memory reference instructions and a set of instructions for manipulating register and memory data on a half-word basis (De75). It is a "conventional" machine in that it does not provide any specialized hardware operations to explicitly support a level-structured operating system (such as those proposed in Be75) or to synchronize the execution of concurrent programs.

Now let us consider the development of a second level starting from the physical system just described. The result of this development will be a "virtual machine" which is supposed to be more convenient to work with than the original machine. As mentioned earlier we will arrive at one of many possible virtual machines, depending on the specific improvement we wish to make in the bottom level. We propose to investigate a virtual machine which eliminates the restriction that the computing system can only be engaged in one task at a given time. This restriction is imposed by the existence of only a single processing unit at the bottom level. If the system contained several processors, we could allow several independent users to have computing work done simultaneously. Therefore we shall specify that the virtual machine appearing at the second level must provide a number of independent processors which can operate concurrently and which can perform the same instructions as the KI10 CPU. We will refer to these processors as "virtual processors" to distinguish them from the physical processor of the bottom level. For the sake of

generality we will not set any upper limit on the number of virtual processors which may be required.

The reader has probably realized that the collection of virtual processors just described is the abstraction created in modern operating systems by multiprogramming a single physical processor. A number of well-known techniques are available for creating such an abstraction and hence implementing a virtual machine of the required type (Lo72). One straightforward approach makes use of a hardware clock to periodically interrupt the CPU, a "scheduler" implemented in software to select the virtual processor to which the CPU shall be assigned next, and a "context switcher", also implemented in software, to restore the CPU to a state consistent with the one which existed when the selected virtual processor was last interrupted. In this approach it is generally true that the virtual processors have no control over the occurrence of an interrupt and the selection of the next virtual processor to run, so these must be considered as essentially random events as far as the virtual processors are concerned. Hence no assumptions can be made about the relative speeds with which the various processors execute their programs.

Because of the widespread acceptance and practice of multiprogramming, we will not dwell further upon the details of implementing an abstract machine which provides a number of independent virtual processors that operate concurrently

(see Lo72, Pr75). We will concern ourselves instead with the question of how the system designer ought to proceed once he has implemented such an abstraction. Before proceeding, however, we ought to recognize an important consequence of the level-structured design approach being followed. We described the virtual machine at the second level as an abstraction of a bottom level containing a single physical processor. It is conceivable that an identical virtual machine could be implemented starting from a different bottom level (i.e., a different set of hardware). For example, we might arrive at the same virtual machine starting from a bottom level containing more than one physical processor. We might even find a case in which a separate physical processor is available for every virtual processor, so that the required virtual machine already exists at the bottom level. Our work from this point on will depend on the characteristics of the specified virtual machine, but will not depend on the nature of the lower level or levels from which it was developed. Hence our results should be applicable to any physical system in which it is possible to implement a virtual machine having the required properties. As a matter of fact, during the course of the experimental work for this investigation a second KI10 CPU was added to the DECsystem-10 being used, providing a true multiprocessor configuration. This addition did not require any change in algorithms developed previously to coordinate the activity of numerous virtual processors, even

though a qualitatively different kind of concurrency could occur among the virtual processors after the addition of the second CPU.

1.4 OUTLINE OF THE DISSERTATION.

The succeeding chapters of this dissertation will describe an investigation into certain problems faced by a system designer at the lower levels of a level-structured operating system whose first abstraction provides a virtual machine of the type described in the previous section. This virtual machine has the following characteristics:

(1) It includes an arbitrarily large number of processors which can all be executing their programs at the same time;

(2) The relative operating speeds of the processors are unspecified and may in fact vary with time;

(3) We assume that the system's primary memory includes a set of storage locations accessible to all the processors, providing them a means to communicate with one another;

(4) The primitive operations available to the processors are the indivisible user-mode machine instructions of the KI10 central processing unit.

By "primitive operations" (at any level) we mean a set of operations having the property that if two or more operations from the set are performed simultaneously, the end result is the same as if the operations were performed one by one in some specific (although not predetermined) order. Requiring certain of the processors' operations to be primitive allows us to legitimately assume that no two of those operations are ever performed at exactly the same time, although there will be cases in which we cannot predict the order in which two operations will be performed by different processors. The "user-mode" instructions of the KI10 CPU are those which can be executed by the program of any user of the system, as opposed to certain instructions which require special privileges.

The problems we will investigate are specifically those which arise from the need to coordinate the activities of interacting processors. Such interaction generally results when several processors share a common resource such as a data base or an input/output device. In Chapter 2 we introduce a pair of rudimentary operations which provide an efficient means for delaying a processor, that is, for causing a processor to suspend and later resume the execution of its program. In Chapter 3 we develop a more advanced pair of operations which allow the processors to enforce mutually exclusive execution within certain "critical sections" of their programs. Then in Chapter 4 we describe a systematic procedure for solving general

processor coordination problems, based on the processor delay and mutual exclusion mechanisms of Chapters 2 and 3. We illustrate the procedure by solving several well-known coordination problems.

Next we take up the problem of finding an implementation for the mutual exclusion mechanism of Chapter 3 which is efficient enough to allow its use at a very low level in a level-structured operating system. In Chapter 5 we formulate a problem called the mastermode/normalmode problem whose solution is an algorithm which implements the mutual exclusion mechanism. The conditions imposed in the statement of the problem are strict enough that only a very efficient algorithm will solve the problem. In Chapter 6 we present an algorithm which solves the mastermode/normalmode problem, and in Chapter 7 we give a rigorous proof that the algorithm is a correct solution.

In Chapter 8 we consider certain processor coordination problems more complex than those solved in Chapter 4, involving simultaneous access to several independent shared resources. We demonstrate that the algorithm described in Chapter 6 can also be used in the solution of these problems. Finally in Chapter 9 we summarize the results of the investigation and suggest several topics for further study.

CHAPTER 2

EFFICIENT SUSPENSION OF PROCESSOR ACTIVITY

2.1 INTRODUCTION.

This chapter begins an attempt to establish cooperative interaction among a group of processors, specifically the virtual processors described in the last chapter. These virtual processors may have arisen from the multiprogramming of one or more physical processing units or may each correspond to one physical processor, depending on the exact physical configuration underlying the virtual processor abstraction. Our primary objective at present is to determine how to make the processors cooperate rather than interfere as they execute separate but related tasks. A secondary objective is to determine the extent to which we can ignore the underlying physical configuration of our computing system and instead view the system in terms of the virtual processor abstraction. Recall that this abstraction gives us an arbitrary number of processors which execute their programs concurrently at indeterminate relative speeds. The primitive operations available to these processors are the indivisible user-mode machine

instructions of the KI10 central processing unit. The only means provided for communication among the processors is a set of memory locations to which all of the processors have access. We are now ready to consider how the activity of such a group of processors might be coordinated.

2.2 THE NEED FOR PROCESSOR DELAY.

One of the fundamental coordination requirements of a group of independent processors is a mechanism which gives them some control over the progress of one another's processing activity (Sa66, Di68a, Ha72, Pr75). An example may serve to clarify this requirement. Suppose that one processor, which we will call the producer, is executing a cyclic program in which it creates and formats packets of data and stores them in a buffer of some kind. Suppose that a second processor, which we will call the consumer, periodically retrieves data packets from the buffer, subjects them to further processing, and transmits them to an output device.

If the above process is to work properly, some kind of coordination between the producer and the consumer is clearly necessary. Without such coordination the producer might occasionally refill the buffer before the consumer has emptied it, resulting in the destruction of the previously produced data packet. Likewise the consumer might

occasionally retrieve a packet before the producer has refilled the buffer, resulting in erroneous output data.

To see how such undesirable behavior might be avoided, let us consider the latter case in more detail. The following question must be addressed: If the consumer reaches that point in its program at which it retrieves a new data packet from the buffer and the producer has not yet deposited a new packet in the buffer, what shall the consumer do? Assuming that the sole function of the consumer is to retrieve and process data packets, there is not much for it to do in this case but wait until the producer has made a new packet available. In other words, the consumer should temporarily cease to execute its program and should resume execution only after the producer has filled the buffer. (We will consider later how such behavior for the consumer can be arranged.)

We will refer to any arrangement which enables a processor to suspend and later resume execution as a "processor delay mechanism". The purpose of the above example was to show that such a delay mechanism is needed to coordinate the activity of independent processors. This is in contrast to situations involving a single processor, when it is generally preferable for the processor to proceed with its execution at the greatest possible speed.

2.3 A POSSIBLE PROCESSOR DELAY MECHANISM.

Consider how a delay mechanism might be implemented using the basic KI10 machine instructions (De75) and the set of shared memory locations available to the virtual processors. Since no instructions are available which give one processor direct control over another's activity, a processor to be delayed must itself perform the operations which result in the delay. Moreover, the only instruction which allows a processor to cease execution (a halt instruction) requires manual intervention before execution can continue, and hence is not suitable for the temporary suspension of execution required by a processor delay mechanism. Thus a processor must continue to run, that is it must keep executing instructions, while waiting for the end of the condition that required it to be suspended.

The above reasoning suggests a possible processor delay mechanism, which we will now describe in terms of the producer-consumer problem of the last section. A shared memory location which initially contains zero will be used for communication between the producer and the consumer. When the producer finishes filling the buffer it sets the shared location to a nonzero value. When the consumer prepares to empty the buffer, it first tests the number in the shared location. If that number is nonzero the consumer proceeds to empty the buffer, but if the shared location contains zero the consumer simply repeats the test. The

consumer thus remains in a tight loop, repeatedly testing the shared location, until that location is set to a nonzero value by the producer. When the consumer finally exits from this loop and finishes emptying the buffer, it sets the shared location back to zero. The producer must perform a similar test prior to filling the buffer, except that it continues to repeat the test until it finds zero in the shared location.

Note that the processor delay mechanism described above requires the processors to remain active even while their progress is suspended. For this reason the kind of activity the processors engage in while testing the shared location has been referred to as "busy waiting" (Di68b). Before spending any more time on this approach, we ought to answer the following question: Is busy waiting an acceptable form of processor delay? Unfortunately we cannot answer this question unequivocally, because the answer depends on the nature of the physical system underlying the abstract system of virtual processors with which we are trying to work. For example, if each virtual processor represents an independent physical processor then busy waiting might be entirely acceptable as a means of processor delay. However, consider the more common case in which the virtual processor abstraction is created by multiprogramming a single physical processor. In this case the physical processor will almost certainly have something better to do than execute a busy wait loop for one of the virtual processors, and hence busy

waiting leads to an intolerable waste of processing power. Since our goal is to arrange for efficient processor coordination regardless of the underlying physical configuration of a computing system, we must abandon the processor delay mechanism suggested earlier and look for one which does not depend on busy waiting.

2.4 A BETTER APPROACH TO PROCESSOR DELAY.

In the preceding section we were led to the use of busy waiting by the fact that our abstract computing system provided no operations which would allow a processor to temporarily stop running. We got into this trouble by trying to ignore the physical system underlying the abstraction while developing a concept (efficient processor delay) whose very meaning depends on the underlying system. We conclude that our first abstraction was inadequate and should have provided a delay mechanism for the processors of the initial virtual machine.

To correct the problem just described we must provide for processor suspension while implementing the abstraction in which the virtual processors first appear. To take a specific example, if we develop a multiprogrammed operating system which will allow a single physical processor to appear as a number of virtual processors operating simultaneously, we should include an explicit processor delay mechanism when implementing the initial abstraction

which creates the virtual processors. That is, we should give the virtual processors explicit delay operations which are (presumably) not present in the instruction set of the physical processor. Starting from some other kind of underlying system, for example one containing several physical processors, we would provide delay operations which operate identically (as far as the virtual processors are concerned) but which would be implemented differently to account for the difference in the underlying configuration. This approach allows us to arrive at identical abstractions from different physical systems, while achieving an acceptable form of processor delay in each system. Having done this, we can proceed with the design of higher levels without worrying about the underlying physical configuration of the system, and can later use the same higher-level design with other configurations.

2.5 THE SLEEP/WAKEUP MECHANISM.

Let us put aside for the moment the question of implementation, and consider the functional appearance of a reasonable processor delay mechanism. Since we intend to provide a mechanism which is functionally the same regardless of our system's physical configuration, it is important that the delay mechanism be as simple as possible. Otherwise we would run the risk of not being able to implement the mechanism efficiently in certain systems,

undesirably restricting the applicability of our general processor coordination scheme.

Because of the requirement for simplicity, we propose to use a rudimentary processor delay mechanism which is similar to Saltzer's BLOCK/WAKEUP mechanism (Sa66) but which only allows a processor to suspend its own execution. The proposed mechanism, which we will refer to as the SLEEP/WAKEUP mechanism, provides two operations known as the SLEEP operation and the WAKEUP(k) operation. A processor which wishes to temporarily suspend its own execution does so by simply performing the SLEEP operation. No other processor is affected by this action. The WAKEUP operation has an argument, the positive integer k, which designates a specific processor. Thus we assume that the processors have been numbered from 1 to N, where N is the total number of virtual processors and may be arbitrarily large. The effect of the WAKEUP operation is to cause a sleeping processor (i.e., one which has suspended itself with the SLEEP operation) to resume its execution. Thus if Processor 13, for example, comes to a SLEEP operation in its program, it does not execute any more instructions until some other processor performs the WAKEUP(13) operation, at which time it resumes execution with the next instruction in its program after the SLEEP operation.

2.6 PENDING WAKEUPS.

The above description of the SLEEP/WAKEUP mechanism does not address the following question: What is the effect of a WAKEUP operation directed at a processor which is not asleep at the time? We could specify that such a WAKEUP operation will simply be ignored, but this approach would make the SLEEP/WAKEUP mechanism very difficult to use. The source of the difficulty is that in the abstract system to which we are proposing to add the SLEEP and WAKEUP operations, the processors have no control over one another's activity. Thus for example if Processor 13 suspends itself with the SLEEP operation and another processor enables it to continue by sending it a wakeup (i.e., by performing a WAKEUP(13) operation), we have no simple way to guarantee that the WAKEUP operation will be performed after rather than before the SLEEP operation. We would like the final result to be the same in either case. Therefore we require that the SLEEP and WAKEUP operations work as follows. If a WAKEUP operation is directed at a processor which is not asleep, its occurrence will be registered and we will say that a "pending wakeup" is in effect for the processor. When the processor later performs the SLEEP operation it will not stop running but will merely cancel the pending wakeup and continue to execute its program.

Although allowing for pending wakeups will undoubtedly make the SLEEP and WAKEUP operations harder to implement, we will tolerate this complication because the SLEEP/WAKEUP mechanism would be hard to use without it. We will not, however, require a processor to "remember" more than one pending wakeup at a time. Thus if a WAKEUP operation is directed at a running processor for which a pending wakeup is already in effect, that WAKEUP operation will be ignored. We will have to take this possibility into account when using the SLEEP/WAKEUP mechanism.

2.7 IMPLEMENTING THE SLEEP/WAKEUP MECHANISM.

As noted earlier, the nature of an efficient SLEEP/WAKEUP implementation depends on the specific physical configuration underlying an abstract collection of virtual processors. If there is one physical processor for each virtual processor, for example, we might use some form of busy waiting to delay a processor or we might require special hardware which would allow a physical processor to be temporarily halted under program control. Saltzer has described a possible implementation for the multiprogrammed single physical processor case (Sa66). This approach uses for each virtual processor a "sleep" bit and a "wakeup waiting" bit which can be examined by the multiprogram monitor when scheduling virtual processors to run and when

determining how to respond to SLEEP and WAKEUP operations performed by the virtual processors.

A SLEEP/WAKEUP mechanism identical to the one we have proposed is provided to system users by the TOPS-10 Monitor of the DECsystem-1077 dual-processor computer system (De74). As noted in Chapter 1, this is the system whose CPU instructions have been chosen as the basic operations for our virtual processors and which will be used later to demonstrate the solution of some processor coordination problems. The SLEEP/WAKEUP implementation used by the TOPS-10 Monitor is similar to the one described above for the single-processor case. Possible complications arising from the presence of two physical processors are eliminated by allowing SLEEP and WAKEUP operations to be performed on only one physical processor, which is called the master processor. If a particular job (as the virtual processors are called) is running on the other processor and attempts to perform a SLEEP or WAKEUP operation, it is not allowed to continue on that processor and is marked by the monitor as being runnable only on the master processor.

We have intentionally not considered implementations of the SLEEP/WAKEUP mechanism in much detail. As noted in Chapter 1, our main objective is to develop techniques for processor coordination starting from a high enough level of abstraction that the specific physical system underlying the abstraction can be disregarded. We assume that the

SLEEP/WAKEUP mechanism can be implemented because we know that it has in fact been implemented (in one form or another) in various computer systems. However, having developed an abstraction in which SLEEP and WAKEUP are available to the virtual processors as primitive operations, we will never again (we hope) have to concern ourselves with the details of the physical system from which the abstraction has been developed.

CHAPTER 3

MUTUAL EXCLUSION

3.1 INTRODUCTION.

As a result of the development in the preceding chapters, we now have a virtual machine to work with in which an arbitrary number of independent processors, as described on page 1-11, can control one another's activity to a limited degree by means of the SLEEP/WAKEUP operations. Since our ultimate goal is to promote the cooperative interaction of such processors, let us determine what kind of cooperation can be achieved with our present abstraction. We will do this by considering a simple problem involving processor interaction, which we will try to solve using the operations available to our assumed virtual processors. (Recall that these operations consist of the indivisible user-mode machine instructions of the KI10 physical processor along with the SLEEP and WAKEUP operations described in the last chapter.) Our consideration of this example problem will lead us to a new processor control mechanism which permits the virtual processors to enforce mutual exclusion among given sections of their programs.

3.2 A SIMPLE COORDINATION PROBLEM.

It is clear that a problem in which processor interaction is a factor must involve at least two processors, so we will suppose that exactly two of our computing system's arbitrary number of processors are active at the time under consideration. For convenience we will refer to the active processors as Processor P and Processor Q. Suppose that Processor P is engaged in a cyclic process in which it periodically updates the information contained in a data file of some kind (perhaps one stored on a magnetic disk unit), and suppose that Processor Q is periodically examining the information in this same file. Finally, suppose that we do not want Q to examine the file while P is updating it. The coordination problem is then to prevent both processors from accessing the file at the same time.

To solve this problem requires some kind of communication between the processors. Our virtual machine provides a medium for such communication in the form of a segment of primary storage (i.e., core memory) to which all processors have access. Hence our first attempt at a solution might involve the use of particular shared storage locations as flags with which the processors would signal one another. For example, Processor P could set a shared storage location to a nonzero value just before starting to update the file, and set that location back to zero after

finishing its update operation. Then Processor Q could examine the same shared location prior to reading the file. If Q found zero in the shared location it could begin reading without conflict, but if it found a nonzero value it would have to wait for P to finish with the file. Q could use the SLEEP operation to be sure of doing its waiting efficiently, in the sense discussed in Chapter 2. However we would not want Q to sleep forever, and thus would require P to perform a WAKEUP(Q) operation after finishing with the file. Another shared location could be used by Q as a flag to signal P to perform the required WAKEUP operation. A second set of such flags would be needed to handle the reverse situation in which Q is already reading the file when P wants to update it.

Unfortunately, a solution of the form just described would be a dismal failure. The trouble is that each processor must perform several distinct operations when preparing to use or finishing with the file, and we have no control over the order in which the processors perform their operations relative to each other. For example, suppose Processor P tests the flag indicating that Q is reading the file, finds it to be zero, and thus determines to begin updating the file. But suppose that before P is able to set the flag which indicates it is updating the file, Q tests that flag and, finding it to be zero, determines to read the file at once. The result would be simultaneous reading and updating of the file, a violation of the required behavior.

There are other ways the solution can fail. For example, suppose that Q finds the file being updated and hence must perform the SLEEP operation to delay itself. But suppose that before Q sets the flag which will signal P to perform the WAKEUP(Q) operation, P finishes updating the file, tests that flag, and finds that no wakeup is needed. Then Q will go to sleep and will not wake up even when the file is no longer being updated. In fact if P does not update the file again later, Q will never wake up.

3.3 THE NEED FOR MUTUAL EXCLUSION.

It is conceivable that we might find a clever sequence of flag manipulations and SLEEP/WAKEUP operations with which the processors could avoid the pitfalls of the previous section. It is not clear, however, how we should go about finding such a sequence of operations, or even how we can be sure that the sequence we find really works properly. Furthermore, we certainly do not want to look for such a clever solution each time we have a new processor coordination problem to solve. It would be much better to have a simple and efficient mechanism whereby a processor could perform a series of operations with no possibility of intervening operations by other processors.

Actually, not all intervening operations would have to be excluded, just those which involve interaction between the processor in question and other processors. It appears

that what we need is a way to identify certain sequences of operations in the program of each processor which will be executed by only one processor at a time. Dijkstra named such program sequences "critical sections" and pointed out their usefulness in solving coordination problems (Di65). If a given processor is performing operations which have been identified as belonging to a critical section, all other processors are excluded from their critical sections. Thus a mechanism which permits identification of critical sections is referred to as a "mutual exclusion" mechanism for the processors. We will show in Chapter 4 that a mutual exclusion mechanism can be used in conjunction with a processor delay mechanism (such as that provided by the SLEEP/WAKEUP operations) to solve processor coordination problems in a logical and systematic manner.

3.4 THE MASTERMODE/NORMALMODE MECHANISM.

We will now describe a simple mutual exclusion mechanism. By "simple" we mean that the mechanism allows critical sections to be identified in a simple way, and not that we expect the actual implementation of the mechanism to be a simple matter. The mechanism takes the form of a new set of operations which we intend to make available to the processors and which we wish the processors to treat as part of their basic set of primitive operations. In essence, we are proposing to develop from our present abstract computing

system a higher-level abstraction in which explicit operations are available for enforcing mutual exclusion.

As a first step toward the next level of abstraction, let us suppose that the processors in our computing system have two distinct states or modes of operation, which we designate as normalmode and mastermode, and that all processors are initially in the normalmode state. The significance of these states is that while any number of processors can be in the normalmode state simultaneously, at most one processor will be allowed to be in the mastermode state at any given time. Let us also suppose that the processors have available two operations, the MASTERMODE and NORMALMODE operations, for changing from one state to the other. Thus if a processor is in normalmode, the state in which all processors are presumed to begin their execution, and the processor wishes to enter mastermode (i.e., to change its state to the mastermode state), it does so simply by executing the MASTERMODE operation. When the processor wishes to return to normalmode it does so by executing the NORMALMODE operation.

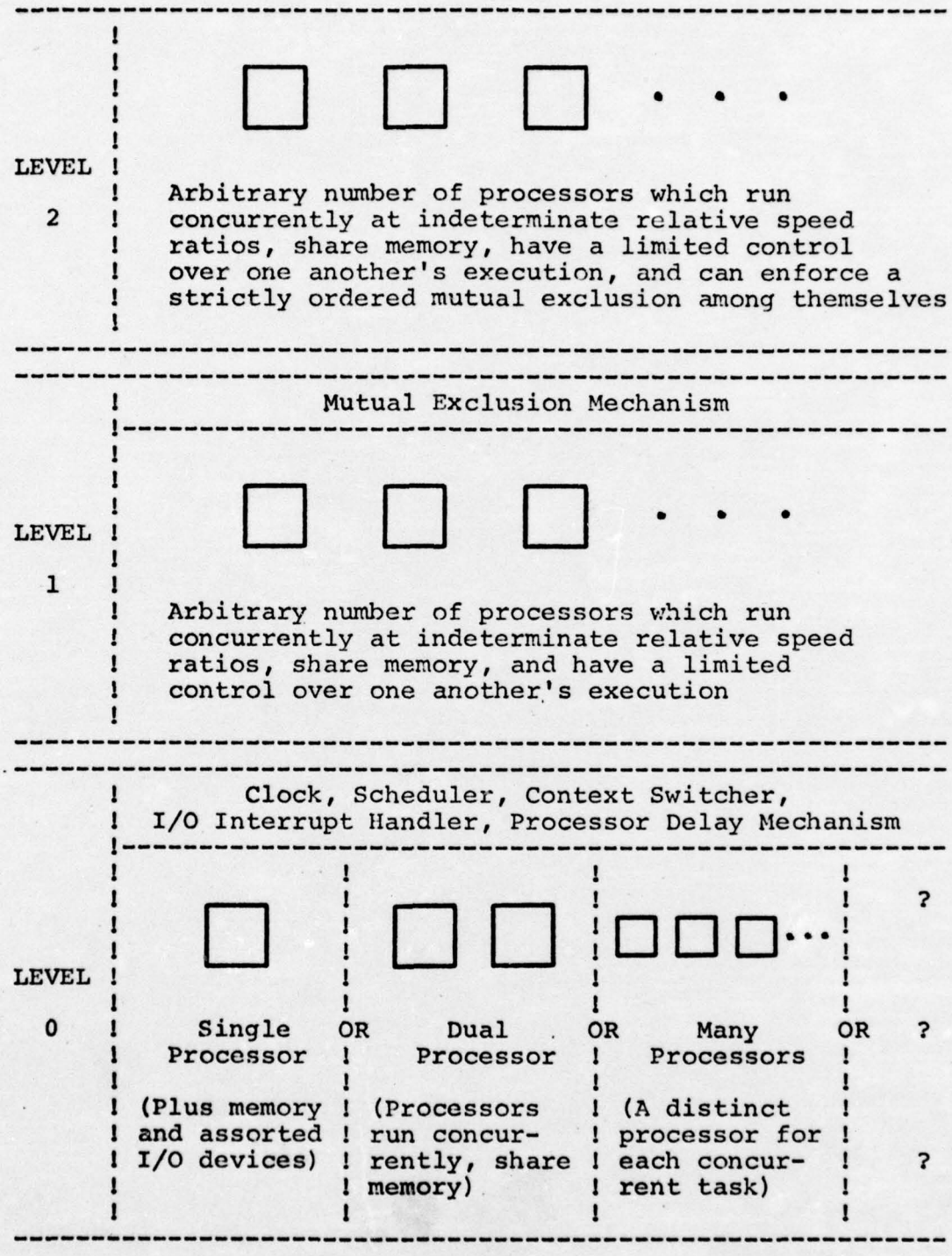
Since only one processor can be in mastermode at a given time, the operations just described provide a simple way to define critical sections and to enforce mutual exclusion. We merely begin each critical section with a MASTERMODE operation and end it with a NORMALMODE operation. This seemingly simple approach may in fact be quite

complicated, but its complications are hidden in the implementation of the MASTERMODE/NORMALMODE operations. For the time being we will assume that these operations can indeed be implemented using the primitive operations available to the lower-level virtual processors described in Chapters 1 and 2.

The diagram on the next page shows the level-structured system which has been developed up to this point. Several possibilities are indicated for the underlying physical system at Level 0. Level 1 represents the virtual machine described in Chapter 2. It is developed from Level 0 by implementing a collection of virtual processors which can perform the SLEEP and WAKEUP operations in addition to the machine instructions of the underlying physical processor(s). A new virtual machine at Level 2 is developed from Level 1 by implementing the MASTERMODE/NORMALMODE mechanism. This implementation, which is carried out on the Level 1 virtual machine, is considered in great detail in Chapters 5, 6, and 7.

Certain questions about the MASTERMODE/NORMALMODE mechanism have not yet been answered. For example, what happens to a processor which performs the MASTERMODE operation when another processor is already in mastermode? Clearly the former processor must be delayed, that is its execution must be temporarily suspended, until the latter processor performs the NORMALMODE operation, so that both

Fig. 3-1. Development of level-structured system. Page 3-8



processors will not be in mastermode at the same time. This brings up another question. If one processor is in mastermode and several others are waiting to enter mastermode (i.e., have been delayed while performing the MASTERMODE operation), which processor will be allowed to enter mastermode next? The MASTERMODE/NORMALMODE operations might be implemented in such a way that no definite choice is made, so that any of the waiting processors might be next to enter mastermode. If this approach were taken, however, a particularly unlucky processor might be delayed indefinitely. We would rather be able to guarantee that any processor which performs the MASTERMODE operation will eventually be allowed to enter mastermode. A simple way to do this is to specify that when several processors are waiting to enter mastermode, the one allowed to enter next shall be the processor which has been waiting longest. In addition to preventing indefinite blocking of a particular processor, requiring the operations to work this way will also simplify the solution of certain processor coordination problems which demand strict first-come-first-served access to a shared resource.

In view of the above considerations, we shall require that processors be allowed to enter mastermode in the precise order in which they perform the MASTERMODE operation. Of course, it remains to be seen whether we can devise an efficient implementation having this property. But before we consider how to implement the MASTERMODE and

NORMALMODE operations using lower-level operations already in existence, we would like to show that such a mutual exclusion mechanism is worth implementing. Therefore in the next chapter we will describe a fairly systematic way to solve processor coordination problems using the MASTERMODE and NORMALMODE operations along with the SLEEP and WAKEUP operations introduced in Chapter 2.

CHAPTER 4

A SYSTEMATIC APPROACH TO PROCESSOR SYNCHRONIZATION

4.1 INTRODUCTION.

In the preceding chapters we have developed a computer system abstraction in which a number of independent processors can control one another's activity to a limited degree by means of the SLEEP/WAKEUP operations and can enforce a strictly-ordered mutual exclusion among themselves by means of the MASTERMODE/NORMALMODE operations. Before considering how to implement the latter set of operations, we wish to assess their usefulness in solving problems which arise as a result of concurrent activity in a computer system. We will begin by presenting a straightforward approach to the solution of processor synchronization problems based on a concept originated by Dijkstra (Di68c). We will then illustrate this approach by using it to solve several versions of a well-known synchronization problem.

4.2 ALLOCATION OF SHARED RESOURCES.

4.2.1 General Considerations.

There are certain situations in which two processors can operate concurrently without any possibility of interference even in the absence of an explicit synchronizing mechanism. In general such situations arise when the processors do not engage in any kind of sharing. Such situations are rare, however, and we usually can expect the relationship between two processors to involve sharing of some kind. For example, processors engaged in related tasks must often share a common data base. Even when their tasks are unrelated, processors usually must share various resources provided by the computing system such as memory banks, mass storage units, and input/output devices. By broadening the idea of a "resource" to include everything that might be shared among a group of processors, we can formulate virtually all the coordination needs of the processors as problems in the allocation of shared resources. This is the viewpoint from which we will consider problems of processor synchronization in the present chapter.

Now let us consider the activity of a processor which is sharing some resource with one or more other processors. We will assume that certain periods in this activity can be identified during which the processor is making active use of the shared resource, and that except in these periods the

processor engages only in private tasks free from interaction with other processors. A processor doing this kind of sharing has two main responsibilities. First, it must not use the resource at the wrong time. Second, it must not prevent another processor from using the resource at the right time. Whether it is right or wrong at a given time for a processor to use the resource is determined by the specific resource allocation problem being solved. In fact we generally define a resource allocation problem by specifying the conditions under which a given processor or class of processors may use the shared resource.

Suppose that the processor under consideration has been programmed so that it never uses the shared resource at the wrong time. This implies that before using the resource, the processor examines the current state (i.e., condition or status) of the resource and the other processors and takes steps to delay itself if the time is not yet right. (Recall that the operations available on our assumed virtual machine allow a given processor to be delayed only by voluntary action on the part of that processor.) Thus the processor must perform certain actions in order to gain access to the shared resource. When the processor has finished using the resource, it must make this fact known so that other processors can correctly assess the state of the system. Based on this reasoning we give the following generalized program to represent the activity of a processor which is making periodic use of a shared resource:

START: perform private tasks;
gain access to the resource;
use the shared resource;
relinquish access to the resource;
go back to START.

The "private tasks" mentioned above are those which do not require any coordinating activity among the processors. Note that the processors are sharing a single resource. More complicated cases, for example the simultaneous sharing of several independent resources, are considered in Chapter 8.

4.2.2 Gaining Access to the Shared Resource.

We will now consider in greater detail the actions performed by a processor which is preparing to use a shared resource. We have already noted that such a processor must examine the current state of the system (which includes the shared resource and all of the processors) to determine whether it can proceed. This implies that information about the state of the system is available to all processors, a condition we can satisfy by requiring the state information to be kept in a segment of shared memory. If a processor, after inspecting this information, determines that it may now use the resource, it must modify the state information before proceeding so that other processors will recognize that the given processor is using the resource.

This necessary updating of the state information leads to a potential problem. If a processor has determined that it can use the shared resource but has not yet modified the state information to reflect that it is doing so, another processor may examine the outdated state information and decide wrongly that it can also use the resource. Thus if we permit several processors to inspect and modify the state information concurrently, we will be confronted with chaos.

Fortunately, we have available a means to eliminate the problem just described. We simply require that a processor preparing to examine the state information enter mastermode before doing so, and not return to normalmode until it has updated the state information to reflect the course of action it has decided to follow. Thus the activities of examining the state of the system, making a decision based on that examination, and updating the state information to reflect that decision are all combined into one primitive operation through the use of the MASTERMODE/NORMALMODE operations.

Now let us consider the case in which a processor must be delayed in using the shared resource. Suppose a processor has entered mastermode, examined the state of the system, and determined that the right conditions do not presently exist for it to use the shared resource. Then after updating the state information to indicate that it wants to use the shared resource but is not yet able to, the

processor must take some action to delay its progress until the right conditions do exist. To accomplish this delay efficiently, we would like to use the SLEEP/WAKEUP mechanism described in Chapter 2. That is, we want the processor in the given situation to delay itself by performing the SLEEP operation, counting on a later wakeup by another processor at the appropriate time.

If the processor is still in mastermode when it performs the SLEEP operation, other processors will be blocked from examining the state of the system until the given processor wakes up and returns to normalmode. This is unacceptable, because there may be other processors which could use the shared resource at once under the existing conditions. Thus the processor must return to normalmode before performing the SLEEP operation.

Now we must allow for the fact that sometimes the processor will be able to use the shared resource at once, and hence should not delay itself. We will do this by requiring the processor to perform the SLEEP operation conditionally, based on the value of a private variable which the processor will set while in mastermode and test after returning to normalmode. In particular, we will establish for each processor a private Boolean variable named MUST.WAIT which the processor will set true when desiring to delay itself and will set false when able to use the shared resource at once. After returning to normalmode

the processor will perform the SLEEP operation or not, depending on whether MUST.WAIT is true or false.

The approach just described for gaining access to a shared resource is summarized in the following program:

```
START:  perform private tasks;
        MASTERMODE;
            examine current state of system;
            if resource cannot be used now, update
               state information to show this pro-
               cessor waiting to use the resource
               and set MUST.WAIT true;
            otherwise, update state information
               to show this processor using the
               resource and set MUST.WAIT false;
        NORMALMODE;
        if MUST.WAIT then SLEEP;
        use the shared resource;
        relinquish access to the resource;
        go back to START.
```

Comparing the above program with the general program on page 4-4, it is clear that the program action originally described as "gain access to the resource" is now being accomplished by the program steps from the MASTERMODE operation down through the conditional SLEEP operation. If a processor sets MUST.WAIT true while in mastermode, it performs the SLEEP operation after returning to normalmode

and its progress comes to a halt. Its fate, which is not yet clear, will be revealed in the next section.

4.2.3 Relinquishing Access to the Shared Resource.

Now let us consider the responsibilities of a processor which has just finished using the shared resource. For one thing, it must update the state information to indicate that it is no longer using the resource. In addition, it is possible that certain processors which are asleep waiting to use the resource should now be allowed to do so. If this is the case, the processor which has just finished with the resource is responsible for waking up those processors. As before, the processor must be in mastermode while examining and modifying the system state information. This approach leads to the program on the next page.

```
START:  perform private tasks;
        gain access to the resource;
        use the shared resource;
        MASTERMODE;
            update state information to show this
               processor not using the resource;
            examine current state of system;
            if some processor, say Processor M,
               is waiting to use the resource and
               now qualifies to do so, perform
               WAKEUP(M) and update state infor-
               mation to show Processor M using
               the resource;
            repeat last two steps until there are
               no waiting processors which qualify
               to use the resource;
        NORMALMODE;
        go back to START.
```

In this program the steps from the MASTERMODE operation down through the NORMALMODE operation take the place of the "relinquish access to the resource" step in the general program on page 4-4.

It is now clear that a processor which put itself to sleep while trying to gain access to the shared resource will continue to sleep until it is awakened by a processor which has just finished using the resource. Of course, the

processor trying to gain access might receive the wakeup before it even performs the SLEEP operation, in which case it will complete the SLEEP operation without delay. (Refer to the discussion of "pending wakeups" in Chapter 2.) We have assumed that a waiting processor only becomes eligible to use the resource as a result of some other processor finishing with the resource. Conceivably a resource allocation problem could arise in which a waiting processor would become eligible as a result of some other processor attempting to gain access to the resource. Such a problem could be solved by an obvious extension of the above scheme in which each processor would make a test while gaining access to the resource to see if it should wake up any other processors.

4.2.4 Procedure for Solving Allocation Problems.

In the last two sections we have described the actions required of the processors provided by our virtual machine when they engage in resource sharing. From this description we derive the following procedure for solving a stated resource allocation problem.

(1) Devise a strategy for each processor or class of processors. This strategy has two parts which specify respectively how the processor will gain and relinquish access to the shared resource. The strategy is determined by answering the following questions:

(a) When a given processor wants to use the shared resource, under what conditions should it be allowed to do so at once and under what conditions should it be delayed?

(b) When a given processor has finished using the resource, under what conditions should it release another processor which is waiting to use the resource?

(2) Decide what information about the state of the system is needed by each processor in order to carry out its strategy. This decision establishes the items to be included in the data base which will be maintained in the shared memory segment accessible to all processors.

(3) Develop a program for each processor incorporating its strategy into the pattern of the general programs listed on pages 4-7 and 4-9.

In addition to providing a systematic approach to the solution of resource allocation problems, the above procedure also simplifies the task of verifying the correctness of a solution. It does so by allowing the proof of correctness to be divided into the separate tasks of verifying that the strategy selected for a given processor leads to the correct behavior of that processor, and verifying that the program for a given processor correctly implements its strategy. It should be noted that a formal procedure for devising a strategy has not been given, and

hence the strategy developed in a given problem is not likely to be unique. The procedure requires that a strategy be found for each processor which will lead to the behavior specified in the problem statement, but does not require that there be only one such strategy.

4.3 CONCURRENT CONTROL WITH READERS AND WRITERS.

4.3.1 Problem Description.

In order to illustrate the procedure described in the last section, we will now solve several example problems. We will restrict our attention to a family of resource allocation problems referred to as the problems of concurrent control with readers and writers, or simply reader/writer problems. The reader/writer problems were first classified by Courtois, Heymans, and Parnas (Co71), who solved two of the problems using the relatively high-level P and V synchronizing operations of Dijkstra (Di68b). Since that time these problems, as well as other members of the same family of problems, have been used by a number of authors as examples and test cases in various studies of processor coordination (Br72, Le72, Li73, Ho74, Pr75).

The reader/writer problems may be described briefly as follows. A group of processors is divided into two classes, called readers and writers. The processors compete for

access to single shared resource, which we may think of as a data file of some kind. Any number of readers may be using the resource (i.e., reading the file) simultaneously, but only one writer may use the resource (i.e., update or supersede the file) at a given time. Thus at any particular time the users of the resource must consist of either no processors, an arbitrary number of readers, or a single writer.

The above requirement is the "basic reader/writer condition" which must be met in all of the reader/writer problems. The individual members of this family of problems are distinguished by "access rules" which specify the order in which access to the resource will be granted when not all requests for access can be satisfied at once. We will consider three particular problems. We identify two of them as the "immediate access problem" and the "writer preference problem". These problems were originally identified as Problem 1 and Problem 2 respectively by Courtois, Heymans, and Parnas, and were called the "weak reader preference problem" and the "writer preference problem" by Presser (Pr75). The third problem we will consider, called the "request-ordered access problem", has not previously appeared in the literature to the author's knowledge.

4.3.2 Statement of the Immediate Access Problem.

In this reader/writer problem it is required that a processor wanting to use the resource be granted immediate access unless the basic reader/writer condition would be violated by such access. In particular, if a reader is reading and a writer is waiting to write, then a request to read by a second reader will be satisfied at once. If several processors are waiting which cannot be allowed simultaneous access, no specific ordering of subsequent accesses to the resource is demanded in this problem. Thus if a writer is writing and several readers and writers are waiting, access to the resource might be granted next to any of the writers or to all of the readers, regardless of the order in which the processors requested to read or write.

Note that in the immediate access problem it is possible for the resource to be kept in continuous use by a stream of readers, in which case any writers wishing to use the resource will be forced to wait indefinitely. The occurrence of such indefinite blocking of writers cannot be considered a defect of a particular solution, because the possibility of such a condition is inherent in the statement of the problem.

4.3.3 Statement of the Writer Preference Problem.

Now suppose that we want a writer to be able to start writing as soon as possible after requesting to do so. Then we would modify the access rule of the previous problem to require that no reader be granted access to the resource while any writer is writing or waiting to write. This leads to a new reader/writer problem, namely the writer preference problem. A solution to this problem will guarantee that a writer waiting to use the resource will do so ahead of any reader which was not already reading when the writer requested to write.

Note that in the writer preference problem it is possible that the readers may have to wait indefinitely while the resource is used by a continuous stream of writers. As before, this possibility is a direct consequence of the problem statement. Also as in the previous problem, no specific ordering is required among several waiting writers.

4.3.4 Statement of the Request-ordered Access Problem.

In the two reader/writer problems just given, certain processors could be prevented indefinitely from gaining access to the shared resource. We will now describe a new reader/writer problem, the request-ordered access problem,

in which such blocking cannot occur. The access rules for the request-ordered access problem are as follows:

(1) A reader may not read until all writers have finished which requested to write before that reader requested to read.

(2) A writer may not write until all processors have finished which requested to read or write before that writer requested to write.

The essence of these rules is that we must allow processors to use the shared resource in the exact order in which they request to do so. Of course the basic condition that access to the resource is granted jointly to the readers and individually to the writers has not changed. As an example, suppose that a writer is writing and several readers and writers are waiting. If the longest-waiting processor is a writer, it must be granted exclusive access to the resource when the present writer finishes. If the longest-waiting processor is a reader, then all readers which have been waiting longer than the longest-waiting writer must be allowed to start reading when the present writer finishes.

In the request-ordered access problem, a processor never uses the resource ahead of another processor which made an earlier request for access. Thus no processor can be denied access indefinitely. (We are assuming, as we have

all along, that a processor using the resource eventually relinquishes it.) Using the resource allocation problem solving procedure discussed earlier the request-ordered access problem is the easiest to solve of the three problems described, and its solution will be given first.

4.4 SOLUTION OF THE REQUEST-ORDERED ACCESS PROBLEM.

4.4.1 Processor Strategies.

We know that processors requesting to use the resource will sometimes have to wait, and that the order in which requests occur is significant. Therefore we propose to provide a "waiting line" which processors can join (at the end) when unable to gain immediate access to the resource. Following the procedure on page 4-10, we now determine strategies for the readers and writers, based on the statement of the request-ordered access problem.

Reader Strategy

(1) If a writer is writing or the line is not empty, join the line and wait to read. Otherwise begin reading at once.

(2) After finishing reading, if no other readers are still reading and the line is not empty, allow the processor at the head of the line (which must be a writer) to leave the line and begin writing.

Writer Strategy

(1) If either reading or writing is going on, join the line and wait to write. Otherwise, begin writing at once.

(2) After finishing writing, if a writer is at the head of the line allow it to leave the line and begin writing. If a reader is at the head of the line allow it and all the other readers immediately following it in line (if any) to leave the line and begin reading.

As mentioned earlier we do not assert that these are the only correct strategies. The reader should take the time to convince himself that these particular strategies do indeed lead to the behavior required of readers and writers in the request-ordered access problem.

4.4.2 Assignment of Shared Variables.

Now let us consider the information required by the processors to carry out the strategies just given. For one thing, a processor must be able to determine whether any writer is writing at a given time. Thus we propose to include in the shared memory segment a Boolean variable, WRITING, which is true when a writer is using (or has at least been given access to) the shared resource. The writers must be able to tell whether one or more readers are reading, and the readers must be able to tell, after

finishing with the resource, whether any other readers are still using it. Thus we introduce a shared integer variable, READERS, whose value is the number of readers which have access to the resource at a given time.

Finally, a processor must be able to check the waiting line for the presence of other processors, to remove the processor at the head of the line, and to join the line itself. The line has the form of a simple queue (Kn73, pp. 234-238) which can be represented by an integer array of N+2 variables, LINE[-1:N], where N is the total number of readers and writers. (Here we have used the ALGOL notation indicating that LINE is a one-dimensional array whose subscript ranges from -1 to N.) This array is located in the shared memory segment and is used to implement the waiting line as follows. LINE[-1] contains the number of the processor at the tail of the line and LINE[0] the number of the processor at the head of the line. LINE[-1] and LINE[0] both contain zero when the line is empty. For K in the range from 1 to N (i.e., the number of one of the processors), LINE[K] contains the number of the processor which follows Processor K in line, provided that Processor K is in line and is not at the tail of the line. Otherwise the value of LINE[K] is immaterial.

The following procedure is performed by Processor K to join the line: Set LINE[LINE[-1]] equal to K and then set LINE[-1] equal to K. The following procedure is performed

by any processor to remove the processor at the head of the line: If LINE[0] is equal to LINE[-1], then set both LINE[0] and LINE[-1] to zero; otherwise set LINE[0] equal to LINE[LINE[0]].

For these procedures to work correctly, two processors must not try to join the line or remove another processor from the line at the same time. We ensure this by requiring that processors only perform these procedures while in mastermode.

For convenience we will assume that the readers are numbered 1, 2, . . ., M and the writers are numbered M+1, M+2, . . ., N. Then the processor at the head of the line is a writer if and only if LINE[0] is greater than M.

The variables WRITING and READERS and the array LINE[-1:N] constitute the shared state information needed to solve the request-ordered access problem. Initially READERS, LINE[-1], and LINE[0] must be equal to zero and WRITING must be false.

4.4.3 Programs for the Readers and Writers.

We must now develop programs which implement the reader and writer strategies, following the pattern of the general programs on pages 4-7 and 4-9. All readers follow the same strategy and hence may use the same program, the only

distinction being that different readers have different numbers. The reader program is listed below.

```
START:  perform private tasks;
        MASTERMODE;
           if WRITING is true or LINE[0] is nonzero,
              join the line and set MUST.WAIT true;
           otherwise increase READERS by one
              and set MUST.WAIT false;
        NORMALMODE;
        if MUST.WAIT then SLEEP;
        use the shared resource;
        MASTERMODE;
           decrease READERS by one;
           if READERS is zero and LINE[0] is
              nonzero, then set WRITING true,
              perform WAKEUP(LINE[0]), and remove
              processor at head of line;
        NORMALMODE;
        go back to START.
```

The above program should be compared with the pattern programs (pages 4-7 and 4-9) and with the reader strategy (page 4-17). Note that since a reader will be responsible for waking up one writer at most, no iteration is required in the testing done by a reader when relinquishing access to the resource.

The following is the program used by the writers:

```
START: perform private tasks;
       MASTERMODE;
          if WRITING is true or READERS is nonzero,
             join the line and set MUST.WAIT true;
          otherwise set WRITING true and
             set MUST.WAIT false;
       NORMALMODE;
       if MUST.WAIT then SLEEP;
       use the shared resource;
       MASTERMODE;
          set WRITING false;
          if LINE[0] is greater than M, then set
             WRITING true, perform WAKEUP(LINE[0]),
             and remove processor at head of line;
          otherwise, if LINE[0] is greater than
             zero, repeat the following as long as
             LINE[0] remains greater than zero and
             not greater than M:  increase READERS
             by one, perform WAKEUP(LINE[0]), and
             remove processor at head of line;
       NORMALMODE;
       go back to START.
```

Note that a writer may have to wake up several readers when relinquishing access to the shared resource. This results in an iterative step at the corresponding point in the program. The pattern program on page 4-9 allows for such iteration.

4.4.4 ALGOL Implementation of the Programs.

We will now describe the implementation of the above solution in a high-order programming language. This implementation has been developed primarily because it satisfies a desire to see the problem actually run on a computer and because it illustrates the way in which the rather abstract virtual machine of Chapters 1 and 2 is realized by the operating system of a time-shared computer facility.

Actual computer programs for the request-ordered access solution are listed on the next two pages. For simplicity, these programs have been written to allow for a maximum of five readers and five writers, each of which uses the shared resource three times before finishing its execution. The programs are written in DECsystem-10 ALGOL, a dialect of ALGOL-60. The programs call a number of separately compiled procedures which are specified by means of external procedure declarations. The SLEEP, WAKEUP, MASTERMODE, and NORMALMODE procedures perform operations already discussed. We will now briefly describe the operation of the other

```
BEGIN   ! READER PROGRAM FOR REQUEST-ORDERED ACCESS PROBLEM;

  BOOLEAN     MUST.WAIT;
  INTEGER     MYNUM,READERS,WRITING,INDEX,HEAD;
  INTEGER ARRAY     LINE1[-1:10];
  EXTERNAL PROCEDURE     MASTERMODE, NORMALMODE, INITIALIZE,
                         ASSIGN, SET, PUT.IN, REMOVE.FROM,
                         REPORT, PAUSE, SLEEP, WAKEUP;
  EXTERNAL INTEGER PROCEDURE     VALUE.OF, RANDOM;
  EXTERNAL BOOLEAN PROCEDURE     TRULY;

  ! ESTABLISH POINTERS TO SHARED VARIABLES;
  READERS:=21; WRITING:=22;
  FOR INDEX:=-1 UNTIL 10 DO LINE1[INDEX]:=INDEX+24;

  READ(MYNUM); INITIALIZE(MYNUM,0); FOR INDEX:=1 UNTIL 3 DO


  BEGIN ! START OF READ CYCLE;

        PAUSE(500+(5*RANDOM));

    MASTERMODE;
        IF VALUE.OF(LINE1[0])>0 OR TRULY(WRITING)
           THEN BEGIN REPORT(MYNUM,"WAITING TO READ  ");
                      PUT.IN(LINE1,MYNUM);
                      MUST.WAIT:=TRUE;
                END
           ELSE BEGIN ASSIGN(READERS,VALUE.OF(READERS)+1);
                      MUST.WAIT:=FALSE;
                END;
    NORMALMODE;

        IF MUST.WAIT THEN SLEEP;
        REPORT(MYNUM,"STARTING TO READ ");
        PAUSE(500+RANDOM);

    MASTERMODE;
        REPORT(MYNUM,"FINISHED READING ");
        ASSIGN(READERS,VALUE.OF(READERS)-1);
        HEAD:=VALUE.OF(LINE1[0]);
        IF VALUE.OF(READERS)=0 AND HEAD>0 THEN
           BEGIN SET(WRITING,TRUE);
                 WAKEUP(HEAD);
                 REMOVE.FROM(LINE1);
           END;
    NORMALMODE;

  END OF CYCLE;

END
```

```
BEGIN   ! WRITER PROGRAM FOR REQUEST-ORDERED ACCESS PROBLEM;

  BOOLEAN     MUST.WAIT;
  INTEGER     MYNUM,READERS,WRITING,INDEX,HEAD;
  INTEGER ARRAY     LINE1[-1:10];
  EXTERNAL PROCEDURE     MASTERMODE, NORMALMODE, INITIALIZE,
                         ASSIGN, SET, PUT.IN, REMOVE.FROM,
                         REPORT, PAUSE, SLEEP, WAKEUP;
  EXTERNAL INTEGER PROCEDURE     VALUE.OF, RANDOM;
  EXTERNAL BOOLEAN PROCEDURE     TRULY;

  ! ESTABLISH POINTERS TO SHARED VARIABLES;
  READERS:=21; WRITING:=22;
  FOR INDEX:=-1 UNTIL 10 DO LINE1[INDEX]:=INDEX+24;

  READ(MYNUM); INITIALIZE(MYNUM,0); FOR INDEX:=1 UNTIL 3 DO

  BEGIN ! START OF WRITE CYCLE;

        PAUSE(500+(5*RANDOM));

    MASTERMODE;
        IF VALUE.OF(READERS)>0 OR TRULY(WRITING)
           THEN BEGIN REPORT(MYNUM,"WAITING TO WRITE ");
                      PUT.IN(LINE1,MYNUM);
                      MUST.WAIT:=TRUE;
                END
           ELSE BEGIN SET(WRITING,TRUE);
                      MUST.WAIT:=FALSE;
                END;
    NORMALMODE;

        IF MUST.WAIT THEN SLEEP;
        REPORT(MYNUM,"STARTING TO WRITE");
        PAUSE(500+RANDOM);

    MASTERMODE;
        REPORT(MYNUM,"FINISHED WRITING ");
        SET(WRITING,FALSE);
        HEAD:=VALUE.OF(LINE1[0]);
        IF HEAD>5
           THEN BEGIN SET(WRITING,TRUE);
                      WAKEUP(HEAD);
                      REMOVE.FROM(LINE1);
                END
           ELSE WHILE HEAD>0 AND HEAD<6 DO
                BEGIN ASSIGN(READERS,VALUE.OF(READERS)+1);
                      WAKEUP(HEAD);  REMOVE.FROM(LINE1);
                      HEAD:=VALUE.OF(LINE1[0]);
                END;
    NORMALMODE;

  END OF CYCLE;

END
```

external procedures to help the reader understand the programs.

The INITIALIZE procedure is called once near the beginning of each program. Its function is to establish access to the shared memory segment used for communication between processors. INITIALIZE has two integer parameters. The first is the number of the processor making the call, represented in these programs by the integer variable MYNUM. A value for MYNUM is read by a given program as input data prior to the call to INITIALIZE. It is the value read for MYNUM by different copies of the same program running at the same time that distinguishes, for example, between different readers. The second parameter of the INITIALIZE procedure specifies a base address in the shared memory segment. It is used in later programs to allow several groups of processors to make independent use of the MASTERMODE/NORMALMODE operations. This parameter is always zero in the present programs.

The ALGOL language being used does not allow for explicit sharing of variables among independent programs. Therefore we must use procedure calls to read from and write into the shared memory segment. The procedure calls VALUE.OF(location) and TRULY(location), where "location" is an integer, return the value stored at the specified location in the shared segment. VALUE.OF returns an integer value and TRULY returns a Boolean value ("true" or "false").

The procedure call ASSIGN(location, value) stores an integer value at a specified shared location, and the call SET(location, value) does the same thing for a Boolean value. These four ALGOL procedures invoke assembly language routines which actually access the shared segment.

The procedures PUT.IN and REMOVE.FROM are provided to manipulate queues such as the waiting line used in this problem. The procedure call PUT.IN(arrayname,number) is performed by the processor with the given number in order to enter the queue defined by the given array name. The call REMOVE.FROM(arrayname) causes the processor at the front of the specified queue to be deleted from the queue. The operations actually performed by these procedures were described on pages 4-19 and 4-20.

The procedure call PAUSE(m) causes the execution of the calling program to be suspended for m milliseconds. Such calls are used to simulate the time spent by a processor in performing private tasks and in using the shared resource. The length of each pause, in the present programs, is determined by calling the procedure RANDOM, which returns a random integer value uniformly distributed from zero to 1000. The value returned by RANDOM is never initialized, so the random sequence generated by successive calls to RANDOM starts off differently each time the same program is run.

The procedure REPORT allows a processor to report the times at which significant events occur. Each call to REPORT generates one line of output data which includes the number of the processor, an identification of the event being reported, and the time of day at which the procedure call occurred (to the nearest sixtieth of a second). In the present programs a processor makes a report whenever it starts waiting to use the resource, starts using the resource, or finishes using the resource.

On the next page is a listing of output data for an experimental run with four readers (numbered 1, 2, 3, and 4) and two writers (numbered 6 and 7). The programs for these six processors were executed by six independent timesharing jobs which were competing with each other and with the jobs of other users for access to the computing system's two physical processors. About thirty jobs were active at the time of this particular run. To produce the listed output data, the reports from all six processors were merged and then sorted into the proper order based on the time of each report. (The times are reported in kilojiffies since midnight. A jiffy is one-sixtieth of a second.) The reader may verify that the behavior of the readers and writers as reflected by the listed output data satisfies the conditions of the request-ordered access problem.

Fig. 4-3. Reader/writer output data.


```
NO.  4 STARTING TO READ  AT   3096.993
NO.  1 STARTING TO READ  AT   3097.013
NO.  6 WAITING TO WRITE  AT   3097.045
NO.  4 FINISHED READING  AT   3097.077
NO.  1 FINISHED READING  AT   3097.079
NO.  6 STARTING TO WRITE AT   3097.086
NO.  6 FINISHED WRITING  AT   3097.137
NO.  1 STARTING TO READ  AT   3097.287
NO.  4 STARTING TO READ  AT   3097.309
NO.  1 FINISHED READING  AT   3097.330
NO.  3 STARTING TO READ  AT   3097.345
NO.  4 FINISHED READING  AT   3097.348
NO.  2 STARTING TO READ  AT   3097.351
NO.  7 WAITING TO WRITE  AT   3097.372
NO.  3 FINISHED READING  AT   3097.392
NO.  2 FINISHED READING  AT   3097.413
NO.  3 WAITING TO READ   AT   3097.433
NO.  6 WAITING TO WRITE  AT   3097.447
NO.  7 STARTING TO WRITE AT   3097.456
NO.  4 WAITING TO READ   AT   3097.508
NO.  7 FINISHED WRITING  AT   3097.527
NO.  3 STARTING TO READ  AT   3097.703
NO.  1 WAITING TO READ   AT   3097.732
NO.  2 WAITING TO READ   AT   3097.751
NO.  7 WAITING TO WRITE  AT   3097.755
NO.  3 FINISHED READING  AT   3097.779
NO.  6 STARTING TO WRITE AT   3097.841
NO.  6 FINISHED WRITING  AT   3097.876
NO.  4 STARTING TO READ  AT   3097.914
NO.  1 STARTING TO READ  AT   3097.923
NO.  2 STARTING TO READ  AT   3097.929
NO.  1 FINISHED READING  AT   3097.969
NO.  6 WAITING TO WRITE  AT   3097.979
NO.  4 FINISHED READING  AT   3097.993
NO.  3 WAITING TO READ   AT   3098.033
NO.  2 FINISHED READING  AT   3098.035
NO.  7 STARTING TO WRITE AT   3098.114
NO.  7 FINISHED WRITING  AT   3098.168
NO.  6 STARTING TO WRITE AT   3098.245
NO.  7 WAITING TO WRITE  AT   3098.275
NO.  6 FINISHED WRITING  AT   3098.326
NO.  3 STARTING TO READ  AT   3098.344
NO.  2 WAITING TO READ   AT   3098.380
NO.  3 FINISHED READING  AT   3099.222
NO.  7 STARTING TO WRITE AT   3099.274
NO.  7 FINISHED WRITING  AT   3099.341
NO.  2 STARTING TO READ  AT   3099.489
NO.  2 FINISHED READING  AT   3099.524
```

4.5 SOLUTION OF THE WRITER PREFERENCE PROBLEM.

4.5.1 Restatement of the Problem.

Now let us obtain a solution to the writer preference problem, using the same general procedure followed above. Recall that in this problem a reader must not be granted access to the resource if a writer is waiting, even though other readers are currently reading. Thus waiting writers have absolute priority over waiting readers.

The problem statement does not specify any kind of priority among waiting writers. However, our approach to shared resource allocation requires that a particular writer be selected to go next when several are waiting, since our simple WAKEUP operation must be directed at a specific processor. The most natural selection is probably to let the longest-waiting writer use the resource next, so that a first-come-first-served discipline is enforced among the writers. Such a discipline is easy to achieve, and may actually enhance the value of the solution in some cases. At any rate a solution which provides this ordering will not violate any conditions of the problem statement. Therefore we propose to find a solution which does provide first-come-first-served access for writers. This leads to the following access rules for the writer preference problem:

(1) A writer must be allowed to write as soon as possible after it has requested to do so. In particular it must go ahead of any reader which was not already reading when the writer requested to write.

(2) Requests to write by several writers must be satisfied in the exact order in which the requests occurred.

The first rule was the original condition for the writer preference problem. We have added the second rule because it may make the solution more useful and because our method requires that a definite choice be made among waiting writers.

4.5.2 Processor Strategies.

It is clear that both readers and writers may have to wait at times. However any waiting writer has priority over any waiting reader, which suggests that we should provide two separate waiting lines, one for readers and one for writers. Now if several readers are waiting, they must not read until a time when no writers are writing or waiting to write, and at that time they should all start reading. Thus all waiting readers will be released together. For this reason, we will imagine that readers wait in a "waiting room" rather than a line, although we will use the same kind of simple queue as above to implement both the room and the line. Appropriate processor strategies are now rather easy to determine.

Reader Strategy

(1) If a writer is writing or the line is not empty, enter the room and wait to read. Otherwise, begin reading at once.

(2) After finishing reading, if no other readers are still reading and the line is not empty, allow the writer at the head of the line to leave the line and begin writing.

Writer Strategy

(1) If either reading or writing is going on, join the line and wait to write. Otherwise, begin writing at once.

(2) After finishing writing, if the line is not empty allow the writer at the head of the line to leave the line and begin writing. Otherwise, allow all readers in the room (if any) to leave the room and begin reading.

4.5.3 Assignment of Shared Variables.

The system state information required by the processors to carry out the above strategies is the same as in the request-ordered access problem, except that there is now a waiting room to keep track of in addition to the waiting line. Thus we will add an integer array ROOM[-1:M] to the shared data base for this problem. (Recall that the readers are numbered 1, 2, . . ., M.) The procedures used by a

reader to enter the room or remove another processor from the room are identical to the corresponding procedures on pages 4-19 and 4-20. The variables WRITING and READERS and the array LINE[-1:N] will be used just as in the previous problem.

4.5.4 Programs for the Readers and Writers.

The following is the program used by the readers:

```
START:  perform private tasks;
        MASTERMODE;
           if WRITING is true or LINE[0] is nonzero,
              enter the room and set MUST.WAIT true;
           otherwise increase READERS by one
              and set MUST.WAIT false;
        NORMALMODE;
        if MUST.WAIT then SLEEP;
        use the shared resource;
        MASTERMODE;
           decrease READERS by one;
           if READERS is zero and LINE[0] is
              nonzero, then set WRITING true,
              perform WAKEUP(LINE[0]), and remove
              processor at head of line;
        NORMALMODE;
        go back to START.
```

Note that this program is almost identical to the reader program for the request-ordered access problem (page 4-21). The only difference is that a reader enters the room instead of joining the line when forced to wait.

The following is the program used by the writers:

```
START:  perform private tasks;
        MASTERMODE;
           if WRITING is true or READERS is nonzero,
              join the line and set MUST.WAIT true;
           otherwise set WRITING true and
              set MUST.WAIT false;
        NORMALMODE;
        if MUST.WAIT then SLEEP;
        use the shared resource;
        MASTERMODE;
           set WRITING false;
           if LINE[0] is nonzero then set WRITING
              true, perform WAKEUP(LINE[0]), and
              remove processor at head of line;
           otherwise, if ROOM[0] is nonzero,
              repeat the following until ROOM[0]
              equals zero:  increase READERS by
              one, perform WAKEUP(ROOM[0]), and
              remove processor at front of room;
        NORMALMODE;
        go back to START.
```

These programs complete the solution of the writer preference problem.

4.6 SOLUTION OF THE IMMEDIATE ACCESS PROBLEM.

4.6.1 Restatement of the Problem.

As a final illustration of the procedure introduced in this chapter for solving shared resource allocation problems, we will obtain a solution to the immediate access problem. Recall that in this problem readers get access to the resource as soon as they make their requests, unless a writer is actually writing at the time. The processor selected to use the resource next when several are waiting is not specified in the problem statement, but our procedure requires that a definite choice be made. The simplest approach would be to give priority either to readers or to writers. We will use a somewhat more complicated approach, merely to illustrate the possibilities afforded by our general procedure. We will solve a version of the immediate access problem defined by the following access rules:

(1) A processor which is ready to read or write must be permitted to begin at once unless the basic reader/writer condition would be violated thereby.

(2) If several processors are waiting to read or write, the one which has been waiting longest must be permitted to go next.

The first rule above distinguishes the immediate access problem from the other reader/writer problems. We have added the second rule to establish a specific order for releasing waiting processors. An example will demonstrate the consequences of these rules. Suppose a writer is writing and several readers and writers are waiting. If the longest-waiting processor is a reader, then by the second rule it must be allowed to read as soon as the present writer relinquishes the resource. But then, by the first rule, all other waiting readers must be allowed to read. Thus if the longest-waiting processor is a reader, all waiting readers will be given access to the resource when the present writer finishes. If the longest-waiting processor is a writer, on the other hand, it will be given exclusive access when the present writer finishes.

4.6.2 Processor Strategies.

Since waiting readers are sometimes released before longer-waiting writers, it is clear that a single waiting line for both readers and writers will not suffice. Thus we will retain the waiting line and waiting room of the previous problem. The following strategies satisfy the access rules given above.

Reader Strategy

(1) If a writer is presently writing, then either join the line (if there is not already a reader in the line) or enter the room (if there is already a reader in the line). But if no writing is going on, begin reading at once.

(2) After finishing reading, if no other readers are still reading and the line is not empty, allow the processor at the head of the line (which must be a writer) to leave the line and begin writing.

Writer Strategy

(1) If either reading or writing is going on, join the line and wait to write. Otherwise, begin writing at once.

(2) After finishing writing, if there is a writer at the head of the line allow it to leave the line and begin writing. Otherwise, if there is a reader at the head of the line allow it to leave the line and begin reading, and also allow all readers in the room (if any) to leave the room and begin reading.

Note that there can be at most one reader in the line, and it "holds the place" for all waiting readers. When it reaches the head of the line, it and any readers in the waiting room will gain access to the shared resource.

4.6.3 Assignment of Shared Variables.

Compared with the previous problems, some additional information is required by the processors to carry out the above strategy. Specifically, a reader which has requested to read and has been forced to wait must be able to determine whether another reader is already in the line. Using the same shared variables as in the writer preference problem, the needed information can be provided by changing the interpretation of READERS. Before, this integer variable represented the number of readers currently having access to the resource. Suppose we let READERS be equal to the number of readers which are either reading or waiting to read. Then a reader which has to wait because writing is going on can determine whether one or more readers are already waiting by checking READERS. Note that at any time when a processor can inspect the shared variables (i.e., when all other processors are in NORMALMODE), the readers which have requested to read are either all reading or all waiting to read, so the value of READERS represents one or the other at a given time. The Boolean variable WRITING and the arrays LINE[-1:N] and ROOM[-1:M] will be used as in the previous problem.

4.6.4 Programs for the Readers and Writers.

The following is the program used by the readers:

```
START:  perform private tasks;
        MASTERMODE;
            if WRITING is true, then join the
               line if READERS is zero and enter
               the room if READERS is nonzero, and
               in either case set MUST.WAIT true;
            otherwise, set MUST.WAIT false;
            increase READERS by one;
        NORMALMODE;
        if MUST.WAIT then SLEEP;
        use the shared resource;
        MASTERMODE;
            decrease READERS by one;
            if READERS is zero and LINE[0] is
               nonzero, then set WRITING true,
               perform WAKEUP(LINE[0]), and remove
               processor at head of line;
        NORMALMODE;
        go back to START.
```

Note that in this program a reader attempting to gain access to the resource will increment READERS regardless of whether it reads at once or is forced to wait.

The following is the program used by the writers:

```
START:  perform private tasks;
        MASTERMODE;
           if WRITING is true or READERS is nonzero,
              join the line and set MUST.WAIT true;
           otherwise set WRITING true and
              set MUST.WAIT false;
        NORMALMODE;
        if MUST.WAIT then SLEEP;
        use the shared resource;
        MASTERMODE;
           set WRITING false;
           if LINE[0] is greater than M, then set
              WRITING true, perform WAKEUP(LINE[0]),
              and remove processor at head of line;
           otherwise, if LINE[0] is greater than
              zero, perform WAKEUP(LINE[0]), remove
              processor at head of line, and if
              ROOM[0] is nonzero repeat the
              following until ROOM[0] equals zero:
              perform WAKEUP(ROOM[0]) and remove
              processor at front of room;
        NORMALMODE;
        go back to START.
```

Note that in this program a writer does not increment READERS when releasing a waiting reader as it did in the previous writer programs. This is a consequence of our modified use of the variable READERS and reflects the fact that when a waiting reader is released, the total number of readers which are reading or waiting to read does not change.

The two programs above complete the solution of the immediate access problem.

4.7 EVALUATION OF THE SOLUTIONS.

In the preceding sections we have solved three rather complex processor coordination problems using a systematic low-level approach. The approach is "low-level" in the sense that it requires no more complicated synchronizing mechanisms than a rudimentary SLEEP/WAKEUP facility implemented at the same level as the virtual processors themselves, and a simple mutual exclusion capability provided by the MASTERMODE/NORMALMODE operations, whose implementation will be considered in the next chapter.

Shorter programs which solve the reader/writer problems can be obtained using more sophisticated synchronizing primitives, as shown by the solutions discovered by Courtois, Heymans, and Parnas using the P and V operations (Co71). However the discoverers pointed out that these

solutions, though simple in form, were not easily found and required several cycles of correction and simplification. The low-level approach we have used requires the solutions to contain much detail that could be avoided if higher-level operations were available. It has the advantages, however, that a relatively systematic solution procedure is possible and that the solution can easily be given desirable properties which would be difficult to achieve with higher-level operations (e.g., the strict access ordering obtained in the reader/writer problems).

The above discussion of high-level vs. low-level synchronization would seem rather academic to a system designer starting with a bare machine whose hardware made no provision for processor coordination. He would have to implement for himself whatever operations he selected to perform such coordination. Our concentration on low-level operations reflects a concern for efficient implementation and a conviction that the alternative high-level synchronizing operations represent too great an abstraction to be efficiently implemented at the lowest levels of a finely level-structured system. The problem of implementing the MASTERMODE/NORMALMODE operations will be treated in detail in the next three chapters.

CHAPTER 5

THE MASTERMODE/NORMALMODE PROBLEM

5.1 INTRODUCTION.

In the last chapter several versions of a well-known synchronization problem were solved in a fairly systematic manner. The solutions required the use of the SLEEP/WAKEUP operations introduced in Chapter 2. In addition, certain operations were required to enforce mutual exclusion among the processors, namely the MASTERMODE/NORMALMODE operations described in Chapter 3. In the present chapter we take up the problem of implementing the latter set of operations on the virtual machine whose organization was described in Chapters 1 and 2. Recall that this virtual machine is an abstraction of an underlying collection of physical devices whose exact form no longer concerns us. We are concerned instead with the characteristics of the virtual machine itself, which may be summarized as follows:

(1) The machine consists of a number of processors which operate concurrently. No information is available on the relative operating speeds of the different processors, and in fact these speeds must be expected to vary with time. The total number of processors may be arbitrarily large.

(2) Each processor can perform certain operations which will in fact be executed directly by physical devices at the lowest level. These operations are in general a subset of the machine instruction set of the system's physical processing unit or units. Some of these operations manipulate information stored in memory locations, and there exists a set of such memory locations to which all of the processors have common access.

(3) Each processor can also perform the SLEEP and WAKEUP operations, which work as described in Chapter 2. These operations give the processors a limited degree of control over each other's activity.

(4) If two or more processors perform any of the above operations at the same time, the effect is the same as if the operations were done in a specific (though not predetermined) order. That is, the operations mentioned in (2) and (3) above are primitive operations, as defined in Chapter 1. This property allows us to assume that no two operations are ever performed simultaneously, and we will often find it convenient to make this assumption.

5.2 THE CRITICAL SECTION PROBLEM.

As mentioned above, we are now concerned with the problem of implementing the MASTERMODE/NORMALMODE operations. That is, we wish to develop an abstraction of

our present virtual machine to create a new (higher level) virtual machine in which MASTERMODE and NORMALMODE are primitive operations available to every processor. This is a special case of a problem known in the literature as the "critical section" problem. The critical section problem was first posed in 1965 by Dijkstra (Di65) in the following form:

Each of N processors is engaged in a cyclic process. A portion of each cycle is identified as a "critical section", and the processors must be programmed in such a way that at most one process will be in its critical section at a given time. The processors can communicate via a set of shared memory locations, but the only available operations on these locations are reading from and writing into one location at a time. A processor must be able to halt outside its critical section without blocking the progress of other processors. If two or more processors are about to enter their critical sections, one of them must eventually do so, regardless of the sequence in which the processors execute their instructions relative to each other.

Note that the problem statement above does not specify the first-come-first-served entry to critical sections that we require the MASTERMODE operation to provide. Thus a solution to the critical section problem will not necessarily suffice to implement the MASTERMODE/NORMALMODE operations.

Dijkstra gave an algorithm which solved the problem as stated above (Di65). A shortcoming of his solution was that a particular processor might be delayed indefinitely from entering its critical section. (Note that the problem statement does not rule out this possibility.) An improved algorithm was devised by Knuth (Kn66) which prevented the blocking of individual processors by guaranteeing that a given processor could enter its critical section within $2^{N-1}-1$ turns after making its request (a "turn" being the execution of a critical section by any processor). In modifications of Knuth's algorithm, the longest possible waiting period was reduced to 1/2 N(N-1) turns by deBruijn (De67) and to N-1 turns by Eisenberg and McGuire (Ei72). None of these algorithms enforced a first-come-first-served discipline among the competing processors. That is, in these algorithms the processor allowed to go next when several processors were waiting to enter their critical sections might or might not be the one which had been waiting longest.

A remarkable new solution to the critical section problem was published in 1974 by Lamport (La74). Lamport's solution, known as the "bakery algorithm", differs from the earlier algorithms in that it provides a limited kind of first-come-first-served access to critical sections. It is also simpler and easier to understand than the previous solutions. Thus the bakery algorithm deserves further consideration for possible use in implementing the

MASTERMODE/NORMALMODE operations. As we will see, the bakery algorithm has a number of characteristics which make it unsuitable for this purpose. Recognizing these characteristics, however, will enable us to develop the criteria for a satisfactory implementation.

5.3 DESIRED MASTERMODE/NORMALMODE PROPERTIES.

5.3.1 General Considerations.

We are proposing to implement the MASTERMODE/NORMALMODE operations at a very low level. That is, the virtual machine we have to work with provides only a rudimentary SLEEP/WAKEUP capability in addition to the basic machine instructions of the underlying hardware. We may wish to use the MASTERMODE/NORMALMODE operations to implement more sophisticated synchronizing primitives, such as Dijkstra's P and V semaphore operations (Di68b) or one of the more general extensions of these operations (Va72, Pr75). To do this successfully, we must have highly efficient MASTERMODE and NORMALMODE operations. In other words, the "overhead" incurred by the processors when entering and leaving mastermode must be as small as possible. We would be happiest, in fact, if our basic hardware could provide an explicit mutual exclusion function. As is often the case, however, our bottom level (the hardware) is fixed, and we must find an implementation which is efficient in the context of a specific physical processor (namely the

DECsystem-10 KI10 central processor). The search for efficiency will probably be the most notable feature of our efforts to implement the MASTERMODE/NORMALMODE operations.

Of the published solutions to the critical section problem, the bakery algorithm of Lamport comes closest to providing an acceptable implementation of the MASTERMODE/NORMALMODE operations. In the following paragraphs we will develop a set of criteria for a satisfactory implementation based on a consideration of the bakery algorithm. This algorithm is based upon one commonly used in bakeries, in which each customer receives a number upon entering the store and the holder of the lowest number is the next one served. The algorithm itself is listed in Appendix A. Although not essential, an understanding of how the bakery algorithm works will help the reader to appreciate the following discussion.

5.3.2 Busy Waiting.

The greatest shortcoming of the bakery algorithm is its use of "busy waiting" to delay a processor's progress. A processor performs a sequence of logical tests and, if all are passed, gains access to the critical section. If any test is failed, it is immediately repeated, and continues to be repeated until it is passed. Thus the processor remains active continuously even when unable to proceed with useful work. This form of processor delay might be acceptable in a

(rather unusual) multiprocessor environment which could provide an actual hardware processor for each processor of the virtual machine. It is certainly not acceptable in the conventional multiprogramming environment, in which one or more physical processors are time-shared to create the virtual processor abstraction.

Our objection to busy waiting should not be construed as a criticism of the bakery algorithm per se, which is merely following the implicit ground rules of the critical section problem. The bakery algorithm and all previous solutions use busy waiting because no other method of processor delay is provided for in the statement of the problem. In implementing the MASTERMODE/NORMALMODE operations, however, we have available an explicit mechanism to control processor delay, in the form of the SLEEP/WAKEUP operations. Our first criterion for a satisfactory implementation is that busy waiting be eliminated through the use of the SLEEP/WAKEUP mechanism. (In taking this approach we are passing to a lower level the responsibility for achieving a specific form of processor delay which is consistent with the underlying physical configuration of the system.)

5.3.3 Cyclic Sleeping.

One way to minimize the inefficiency of busy waiting is to use "cyclic sleeping". In this approach, a processor which has failed a test that would have allowed it to proceed with its program does not immediately repeat the test. Instead it performs a special form of the SLEEP operation which causes it to wake up automatically after a specified elapsed time. It then repeats the test to see if it can proceed. The advantage over busy waiting is that the processor does not remain continuously active while waiting to proceed.

The primary attraction of cyclic sleeping is that available algorithms which use busy waiting can be adapted by simply replacing each "busy wait" loop with a "cyclic sleep" loop. The method has some significant disadvantages, however. For example, it requires a new and previously unnecessary form of the SLEEP operation. Also, it may involve considerable overhead because the testing of the condition being waited on may have to be repeated many times. (Here it offers a quantitative but not qualitative advantage over busy waiting.) A final disadvantage of cyclic sleeping is that a processor must wait until the end of its latest sleep period to proceed, even after the condition it is waiting on has been satisfied, resulting in a certain amount of unnecessary delay. Because of these

disadvantages we require that cyclic sleeping not be used in implementing the MASTERMODE/NORMALMODE operations.

5.3.4 First-come-first-served Access.

The bakery algorithm is a first-come-first-served method in the following limited sense. A processor which is about to enter its critical section passes two points in the program, which we will call (in order of appearance) A1 and A2, prior to reaching any point at which its progress can be delayed. Points A1 and A2 are each defined by a single primitive operation, so that a definite sequence can be established for the times at which the points are passed by the various processors. If a certain processor passes point A2 before another processor passes point A1, then the former processor will always perform its critical section ahead of the latter processor. This is not the strictest kind of first-come-first-served behavior, because if there is a time when two processors are both between points A1 and A2, we cannot predict the order in which they will perform their critical sections, even if we know the order in which points A1 and A2 were passed by each processor.

Although the behavior just described for the bakery algorithm prevents indefinite blocking of individual processors (which was one objective in requiring first-come-first-served access to mastermode), it is not sufficient for solving those shared resource allocation

problems in which strict first-come-first-served access to a shared resource is demanded by the problem statement. So that such problems will be easy to solve, we require that the MASTERMODE operation have the following property. Each processor which performs the MASTERMODE operation will pass a certain point in its program, which we denote as point A, prior to any possibility of being delayed. Processors must enter mastermode in the precise order in which they passed point A. In other words, to be considered satisfactory the implementation must provide the strictest kind of first-come-first-served access to mastermode.

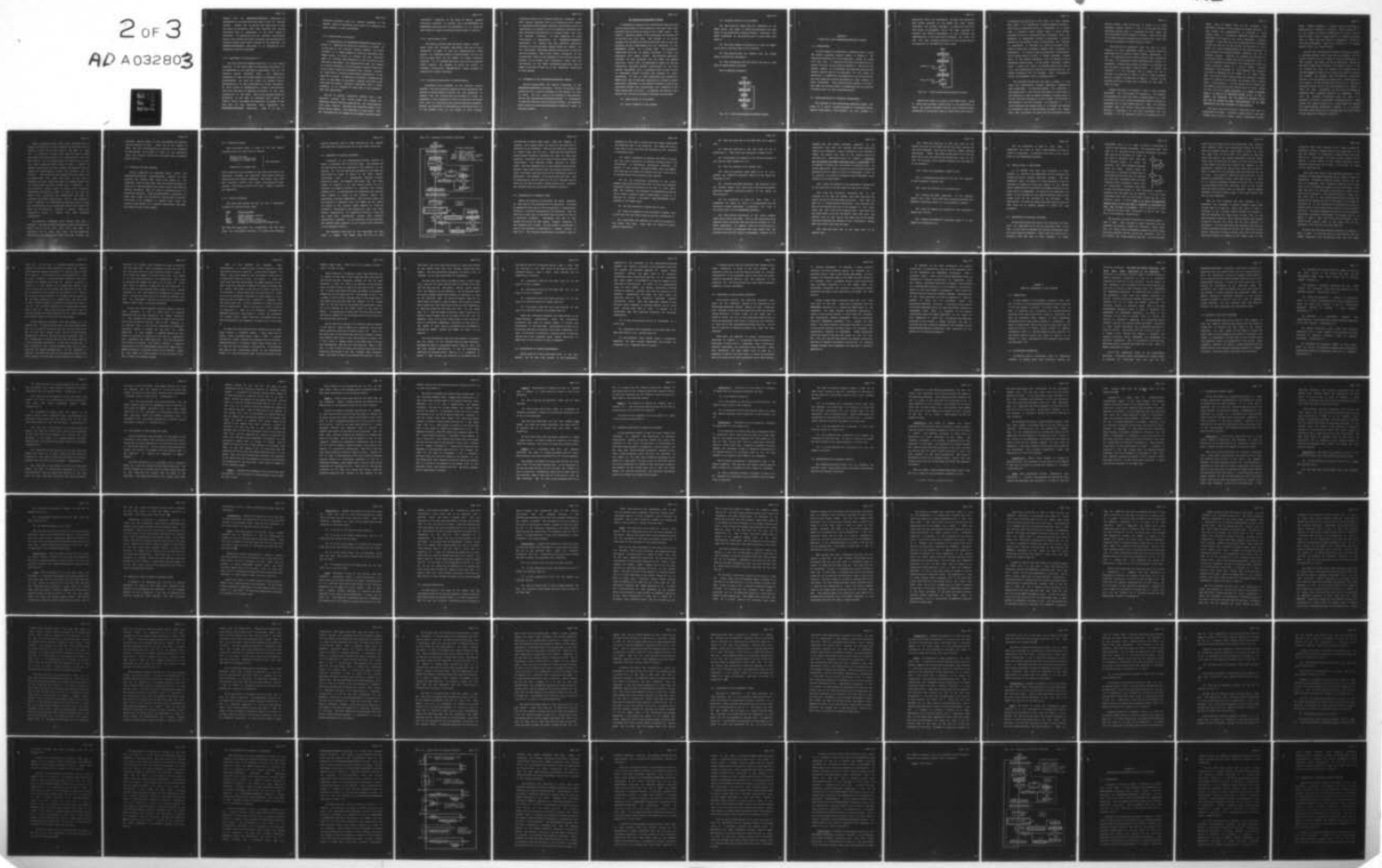
5.3.5 Iterated Testing.

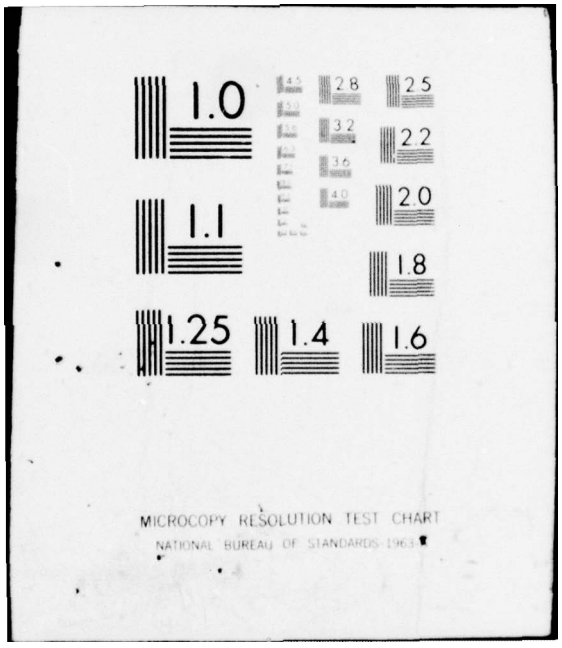
In the bakery algorithm there are two loops in which a processor must perform one or more logical tests associated with another specific processor and which must be repeated until the tests have been made for every processor in the group. Because of our emphasis on efficiency, such iterated testing is rather distasteful. For one thing, we will suffer the overhead normally associated with loops (e.g., the processing required to increment indices and test loop counters). Even more serious, the overhead incurred in entering a critical section will become greater and greater as the total number of interacting processors increases, even though some of the processors may enter their critical sections very infrequently. To eliminate such overhead, we

AD-A032 803 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OHIO SCH--ETC F/G 9/2
EFFICIENT MULTIPLE PROCESSOF COORDINATION.(U)
NOV 76 B E BAKER

UNCLASSIFIED DS/EE/76-1 NL

2 OF 3
AD A032803




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

require that the MASTERMODE/NORMALMODE operations be implemented by an algorithm which makes no use of iterated testing. Another way of stating this requirement is that the amount of processing done when entering and leaving mastermode must be independent of the total number of processors. In conjunction with our previous restrictions against busy waiting and cyclic sleeping, the prohibition of iterated testing means that in essence we are requiring the MASTERMODE/NORMALMODE operations to be implemented by a completely loop-free algorithm.

5.3.6 Dependence of Algorithm on N.

One data item which appears explicitly in the bakery algorithm is the total number of interacting processors, N. Its appearance in the algorithm is unavoidable due to the iterated testing mentioned above. One undesirable consequence is that the maximum number of processors must be predetermined, and once set cannot be changed without reprogramming or at least restarting all of the processors. We would prefer an implementation in which a new processor could join the group of competing processors at any time, irrespective of any prior limit on the maximum number of processors. In the hope of achieving this result, we require that N, the number of processors, not appear in the data base of the algorithm which implements the MASTERMODE/NORMALMODE operations. That is to say, a

particular processor must not require knowledge of the maximum number of processors with which it is competing (or might compete) to enter mastermode.

5.3.7 Shared Memory Allocation.

In implementing the MASTERMODE/NORMALMODE operations a means of communication among the processors is needed. As in the case of the critical section problem, the means provided is a memory segment to which all processors have common access. If we are to achieve the goal mentioned above of allowing new processors to join the group at any time and in any number, our implementation must not depend on a static allocation of shared memory locations made prior to the time at which the processors start running. Instead, we would like to allocate shared memory dynamically, as the need for it arises. In addition to avoiding a predetermined limit on the number of processors, dynamic allocation would also allow the shared locations associated with a given processor to be released for other uses if that processor were ever deactivated.

The use of dynamic allocation implies that a new processor joining the active group will obtain any additional shared locations required for its participation from among whatever locations are currently unused in the shared memory segment. This requires, in particular, that the implementation not depend on the shared locations being

contiguous. Therefore in the hope of making dynamic allocation possible, we specify that the MASTERMODE and NORMALMODE operations will be implemented by an algorithm which does not require contiguous shared memory locations.

5.3.8 Shared Memory Size.

The bakery algorithm uses 2N shared memory locations, there being two locations associated with each of the N processors. In order to rule out implementations which require an inordinate amount of shared memory, we will set 2N as an upper limit on the number of shared locations required. It is intended that this restriction be applied rather loosely. For example we would not reject an otherwise satisfactory implementation simply because it required 2N+1 shared locations.

5.3.9 Permissible Operations on Shared Memory.

According to the statement of the critical section problem, the only primitive operations which processors can perform on shared memory locations are the operations of reading from and writing into one location at a time. An advantage of this limitation is that the resulting solutions will work on practically any existing physical processor. However the limitation is unnecessarily restrictive when, as in the present case, we are trying to find a highly

efficient solution for a specific physical processor. The KI10 central processor has in its instruction set a number of read-modify-write memory reference instructions which are constrained by hardware to act as primitive operations even when performed simultaneously in a system having more than one physical processor. To take advantage of these instructions in implementing the MASTERMODE/NORMALMODE operations, we specify that the primitive operations available to each processor, besides SLEEP and WAKEUP, are the indivisible user-mode machine instructions of the KI10 central processor. This is our single relaxation of the restrictions given in the statement of the critical section problem. It is through this relaxation, along with the availability of the SLEEP and WAKEUP operations, that we expect to be able to meet all of the additional conditions we have imposed.

5.4 STATEMENT OF THE MASTERMODE/NORMALMODE PROBLEM.

We have listed above the desired properties of the MASTERMODE/NORMALMODE operations. We will now collect these properties into a concise statement of the problem we wish to solve. We will refer to the problem in this specific form as the "mastermode/normalmode problem" to distinguish it from the original critical section problem. An algorithm which solves the mastermode/normalmode problem is given in the next chapter.

The Mastermode/Normalmode Problem

N independent processors are concurrently executing the cyclic program shown in the flowchart on the next page, each processor having initially begun at the START point. The relative operating speeds of the processors are unknown and in fact may vary with time. The problem is to devise programs for the ENTRY and EXIT blocks such that at most one processor can be in mastermode (i.e., be executing in the MASTERMODE block) at a given time. The processors communicate via a set of memory locations to which all processors have common access. The primitive operations available to each processor are the SLEEP and WAKEUP operations (which work as described in Chapter 2) and the indivisible user-mode machine instructions of the KI10 central processing unit. A processor must be able to halt in normalmode without blocking the progress of other processors. If two or more processors have reached the ENTRY block, one of them must eventually be allowed to enter mastermode regardless of the relative sequence in which the processors execute their instructions (i.e., deadlock of the whole system must not occur). In addition the solution must satisfy the following previously discussed restrictions:

(1) Busy waiting is not allowed.

(2) Cyclic sleeping is not allowed.

(3) Iterated testing is not allowed.

(4) There must be a point near the beginning of the ENTRY block, say point A, which each processor can reach prior to any possibility of being delayed. Processors must enter mastermode in the precise order in which they reach point A.

(5) The total number of processors, N, must not appear as an item in the data base of the solution.

(6) The solution must not require that the shared memory locations be contiguous.

(7) The N processors must not require the use of more than 2N shared memory locations.

(End of problem statement.)

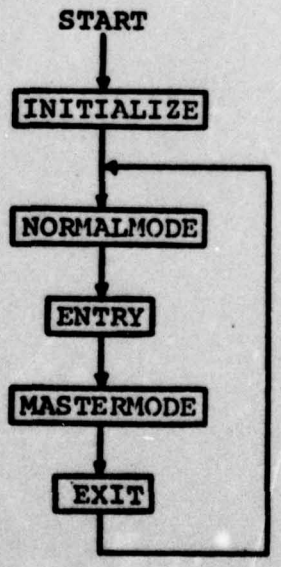


Fig. 5-1. Cyclic mastermode/normalmode program.

CHAPTER 6

A SOLUTION TO THE MASTERMODE/NORMALMODE PROBLEM

6.1 INTRODUCTION.

In this chapter an algorithm is presented which solves the mutual exclusion problem stated in Chapter 5. The algorithm satisfies all of the conditions listed on pages 5-15 and 5-16. A detailed proof that the algorithm is a correct solution will be given in Chapter 7. The aim of the present chapter is to introduce the algorithm and give the reader a general idea of its operation. We will begin by describing a scheme for classifying the interactions which can occur among the various processors as they compete to enter mastermode. This classification scheme, it is felt, will make the solution easier to understand and will provide a starting point for the correctness proof.

6.2 CHARACTERIZATION OF PROCESSOR INTERACTIONS.

The statement of the mastermode/normalmode problem was based on a cyclic program executed by N processors which operate concurrently. The flowchart for this program is

reproduced below for convenience. To solve the problem we must devise programs for the ENTRY and EXIT blocks satisfying the problem statement on pages 5-15 and 5-16. Even though all processors follow the same algorithm in entering and leaving mastermode, we will simplify the implementation of the algorithm by assuming (at least for now) that each processor executes its own private copy of the programs for the ENTRY and EXIT blocks.

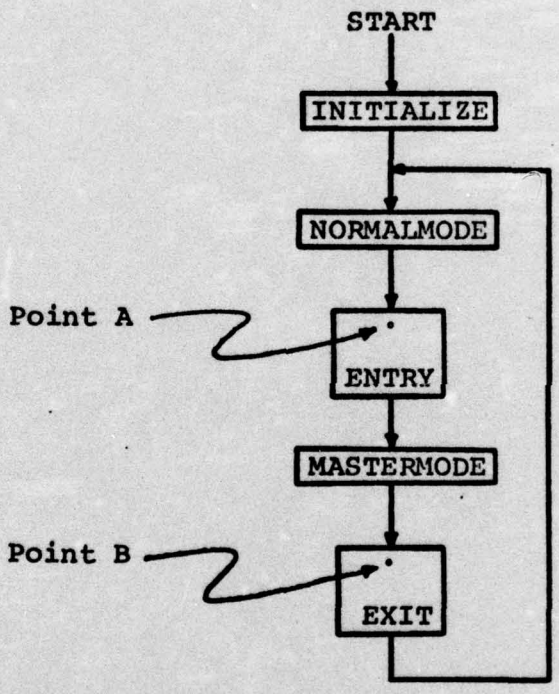


Fig. 6-1. Cyclic mastermode/normalmode program.

Recall that there is a point in the ENTRY block, point A, which each processor reaches prior to any possibility of delay. The problem statement requires that processors enter mastermode in the precise order in which they reach point A.

To eliminate any ambiguity in this order, we will require that point A be defined by a single primitive operation. Since only one processor can be in mastermode at a given time, a processor which reaches point A while another processor is still in mastermode must be delayed until the processor in mastermode returns to normalmode. A precise meaning for the phrases "still in mastermode" and "returns to normalmode" is provided by a point in the EXIT block, point B, which each processor passes while returning to normalmode. If there is just one processor in mastermode, and it passes point B before a second processor reaches point A, the second processor will be allowed to enter mastermode without delay. If the second processor reaches point A while the first is between point A and point B (i.e., has reached point A and has not reached point B), the second processor will be delayed. Thus a processor is "still in mastermode" until it reaches point B.

The following definitions will make it easier to talk about the relationships between processors competing to enter mastermode. If a given processor, say Processor P, has passed point A, and the next processor to reach point A, say Processor Q, does so before Processor P reaches point B, we will say that Processor P is the "leader" of Processor Q, and Processor Q is the "follower" of Processor P. If a third processor now reaches point A, it becomes the follower of Processor Q (not P) and Processor Q becomes its leader. Note that a processor can enter and exit mastermode without

having a leader. This is the case if there is no other processor between points A and B at the time the given processor reaches point A. A processor can also enter and exit mastermode without having a follower. This is the case if no other processor reaches point A during the time that the given processor is between points A and B.

The activity of a processor from the time it first attempts to enter mastermode until it finally returns to normalmode will be called a "pass" through mastermode. We will now characterize the various kinds of pass a processor can make. If a processor enters mastermode and returns to normalmode without having either a leader or a follower, we will say that it has made a "type N" pass. (N stands for "neither".) If the processor had a leader but not a follower, its pass was of "type L", and if it had a follower but not a leader, its pass was of "type F". Finally, if the processor had both a leader and a follower, it has made a "type LF" pass.

Suppose Processor P is making a type F pass through mastermode, and its follower is Processor Q. If P remains in mastermode for some time after Q reaches point A, Q must be delayed so as not to enter mastermode before P returns to normalmode (or at least passes point B). Now, the problem statement does not allow a "busy wait" loop as a means of delaying a processor, so the only legitimate way for Processor Q to be delayed is for it to voluntarily go to

sleep. Thus we suppose that in the situation just described, Processor Q will perform the SLEEP operation. It then becomes the responsibility of Processor P, after passing point B, to perform a WAKEUP(Q) operation so that Processor Q will wake up and enter mastermode. Clearly, Processor P must have some way to determine which processor to wake up, that is, to find out who its follower is. But P cannot do this by checking each of the other processors in turn, because the problem statement prohibits the use of iterated testing. Instead, we suppose that Processor Q, just before going to sleep, has left an indication that it is to be awakened by Processor P (for example by storing its unique processor number in a shared memory location associated with Processor P). This requires that Processor Q be able to determine who its leader is. It seems likely that this can be done, since as soon as Processor Q reaches point A it obtains as its leader the processor which reached point A just before it. We will assume for the moment that the machine instruction which defines point A can be chosen in such a way that a given processor, as a result of passing point A, is able to (1) determine whether or not it has a leader, (2) determine who its leader is, whenever it has one, and (3) arrange for these determinations to be made correctly by the next processor to reach point A.

The type F pass by Processor P described above may be summarized as follows. Processor P passes point A and, upon finding that it has no leader, enters mastermode without

delay. Before Processor P reaches point B, Processor Q passes point A, becoming P's follower. When Q finds that P is its leader, it leaves a message for P to wake it up and goes to sleep. After P passes point B, it reads the message from Q, wakes Q up, and returns to normalmode. Q wakes up and enters mastermode. (Note that Processor Q is making either a type L or a type LF pass.)

We have just described the "normal" interaction between a processor making a type F pass and its follower. We will further classify this as a "type F1" pass to distinguish it from another case which will now be described. Suppose in the above example that Processor P has passed point B and has reached the point of reading the message from its follower before Processor Q has reached the point of leaving the message. We consider this an "abnormal" interaction because it is not expected to happen often. Nevertheless it can happen, and our solution must allow for it. When this situation arises, we will say that Processor P is making a "type F2" pass. Discussion of how it can be handled is postponed until a complete solution is presented. (Note that the abnormal interaction would not occur if a processor, as part of the process of passing point A, could notify its leader to wake it up. Since point A is defined by a single machine instruction, however, it is unlikely that this notification can be included with the other things we are requiring to happen at point A.)

When a processor makes a type L pass, the same kind of abnormal interaction can occur between the processor and its leader, in which case we will refer to the pass as "type L2". In contrast to this is the "type L1" pass, in which the normal interaction occurs (i.e., the follower has left the message for its leader before the leader tries to read it). When a processor makes a type LF pass, a normal or abnormal interaction can occur between the processor and its leader and, independently, between the processor and its follower. Thus there are four kinds of LF pass, which will be denoted L1F1, L1F2, L2F1, and L2F2. To say that a processor has made a type L1F2 pass, for example, means that (1) the processor had both a leader and a follower; (2) the processor, before going to sleep prior to entering mastermode, notified its leader to wake it up, and the leader later recognized the notification and woke the processor up (normal interaction); and (3) the processor, after finishing in mastermode and passing point B, looked for a notification from its follower before the follower had made any notification, requiring a special (and as yet unspecified) action to handle this case (abnormal interaction).

The preceding paragraphs have defined nine ways a processor can pass through mastermode, namely the passes of type N, L1, L2, F1, F2, L1F1, L1F2, L2F1, and L2F2. By specifying one of these nine types, one can completely characterize the interactions which took place between the

processor and its leader (if any) and between the processor and its follower (if any). (Note: We do not intend to imply that this classification is inherent in the statement of the problem. It is based, in fact, on our assumption about the activity which will take place when a processor passes point A.)

6.3 DATA BASE FOR THE SOLUTION.

Before presenting an algorithm which solves the mastermode/normalmode synchronization problem, we will describe the various data items which appear in the algorithm. Some of these data items are shared variables to which all of the processors have common access, and some are private variables, each of which is accessible only to a particular processor. Since the permissible operations on these data items are the machine instructions of the KI10 CPU (De75), it is necessary to specify which items are stored in fast memory (i.e., machine accumulators) and which are stored in main (core) memory.

6.3.1 Shared Variables.

The N processors share a total of N+1 core memory locations which are identified as follows:

Shared flag word	)
Processor 1's wakeup word	)
Processor 2's wakeup word	)
.	) N+1 locations
.	)
.	)
Processor N's wakeup word	)

These locations can be arranged in any order and need not be contiguous, although the algorithm does require that the wakeup words have nonzero addresses. The shared flag word must initially contain zero before any processor starts running. The initial values of the other shared variables are immaterial.

6.3.2 Private Variables.

The local data storage for each of the N processors includes the following data items:

Name	Use or Contents
AC	General-purpose register
J	Index register
MYNUM	This processor's number
FLAG	Address of shared flag word
WAKEUP	Address of this processor's wakeup word

The first two items above are accumulators and the last three are core memory locations. All three core locations

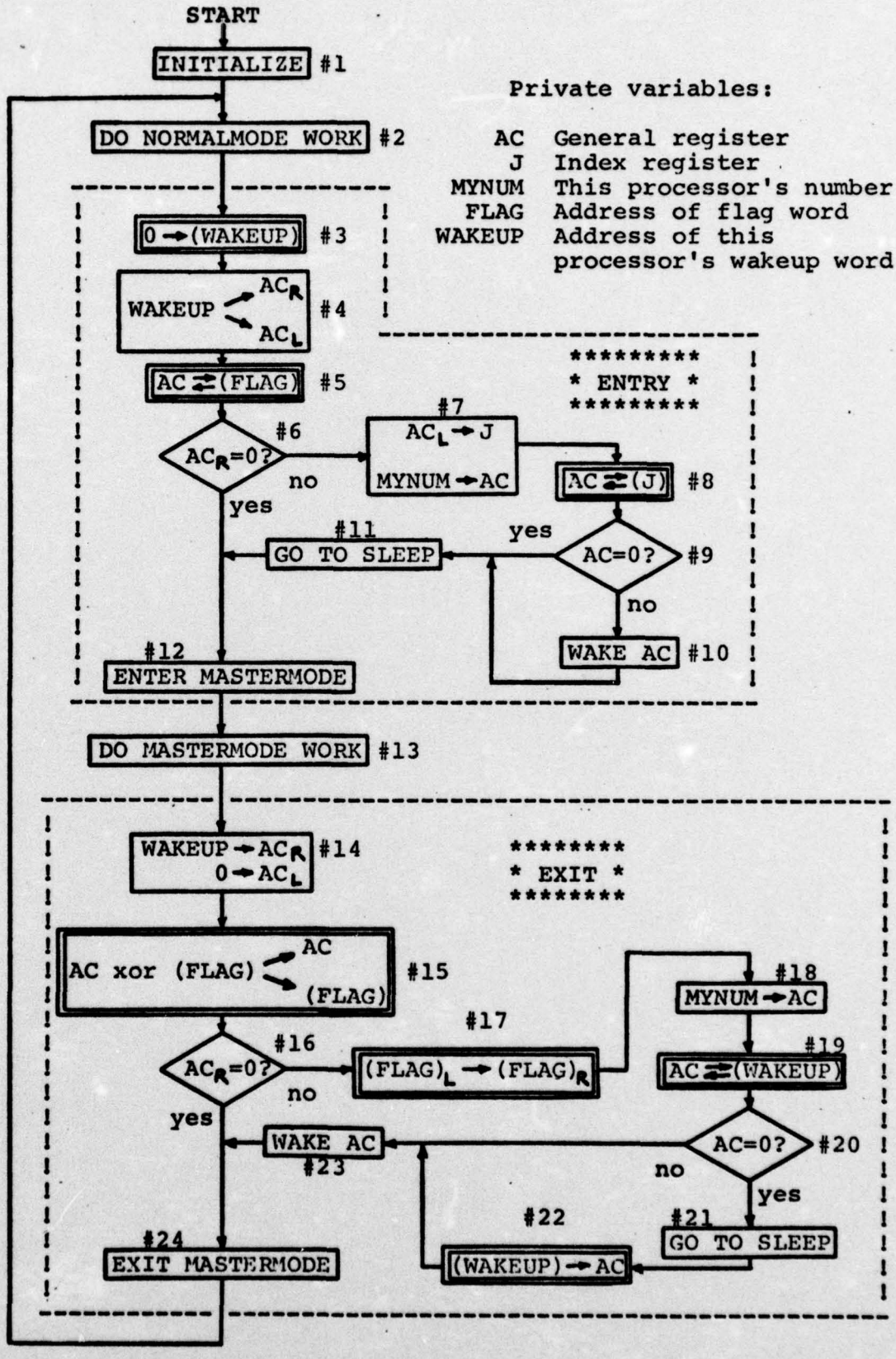
contain constants, that is, their contents are not changed once they have initially been set to the values indicated.

6.4 FLOWCHART OF SOLUTION ALGORITHM.

A solution to the mastermode/normalmode problem is shown in flowchart form on the next page. This diagram has the same form as the one appearing in the statement of the problem, but now the operations performed in the ENTRY and EXIT blocks are shown in detail at the (KI10) machine instruction level. A conventional register transfer notation is used. For example, A→B means that the value stored at location A (or the value A itself, if A is a constant) is stored at location B, replacing the current contents of B. A parenthesized term identifies the location whose address is stored at the location within the parentheses (indirect addressing). Thus A+B→(C) means that the contents of locations A and B are added, and the result is stored at the location whose address is stored at location C. Subscripts L and R denote the left and right halves, respectively, of a specified location or value and are used when representing certain KI10 halfword instructions necessary to the operation of the algorithm. The variable names appearing in the flowchart were described in the preceding section.

For reference, each step of the algorithm has been given a number. The number sign (#) will be used

Fig. 6-2. Flowchart of solution algorithm. Page 6-11



consistently to denote these steps. Thus, for example, #5 standing alone always refers to step #5 of the algorithm as indicated in the flowchart. Some of the steps are enclosed in double boxes (#3, #5, etc.). These are the steps which modify shared variables, and which can thus be considered primitive operations only by virtue of being single machine instructions. The remaining steps, enclosed in single boxes, may each represent one or more machine instructions. However, these steps involve only the private variables of a given processor, and hence may also be considered primitive operations. (For example, if two processors have done step #7 concurrently, the net effect is as if it had been done by first one and then the other, simply because each processor has accessed only its own private storage locations.)

6.5 DESCRIPTION OF FLOWCHART STEPS.

Before the solution is explained, the action performed at each step of the algorithm will be described. It should be emphasized that each of the N processors is independently executing a program represented by the flowchart. Thus the activity at each step is described from the point of view of a particular processor. (Throughout the rest of this chapter and all of the next chapter frequent reference to the flowchart will be made. For the reader's convenience, a copy of the flowchart is provided as a foldout drawing on page 7-57.) The following description of flowchart steps is

presented at this time to ensure that the reader understands the symbology used in the flowchart. The reader may wish to refer back to this list when the operation of the algorithm is discussed later.

#1 - After a processor is started, and before it enters the main cycle, certain of its private variables need to be initialized. In particular, appropriate constants are stored at the private locations MYNUM, FLAG, and WAKEUP. This step does not affect the shared flag word, which had to be cleared before any processors were started.

#2 - All the noncritical processing of the cyclic program takes place within this step, and here the processor presumably spends the majority of its time in each cycle. When the processor has finished its normalmode processing and is ready to request access to mastermode, it does so in practice by calling a procedure named MASTERMODE, which consists of the ENTRY block.

#3 - Set this processor's wakeup word to zero.

#4 - Store the address of this processor's wakeup word in both the left and right halves of private accumulator AC.

#5 - Interchange the contents of AC and the contents of the shared flag word. (This must be done by a single machine instruction.)

#6 - Test the value now in the right half of AC against zero.

#7 - Move the value now in the left half of AC to accumulator J. Then store this processor's number in AC.

#8 - Interchange the contents of AC and the contents of the location whose address is in J.

#9 - Test the contents of AC against zero.

#10 - Wake the processor whose number is in AC (i.e., perform the WAKEUP(M) operation, where M is the number now contained in AC).

#11 - Perform the SLEEP operation. The processor will not proceed beyond this point until some other processor performs a WAKEUP(K) operation, where K is the number of this processor.

#12 - No processing is done at this step. It represents the point at which it is permissible for the processor to enter mastermode. In practice, this step is the return point of the MASTERMODE procedure.

#13 - The critical processing of the cyclic program takes place within this step. (This is the processing which requires mutual exclusion from the critical sections of all other processors.) In general there is no restriction on the kind or amount of processing done here, except that the processor must not halt while in mastermode. However, it is

assumed that the shared variables appearing in the algorithm's data base are altered only where explicitly shown in the flowchart. Hence a processor is not allowed to modify the shared flag word or any wakeup word while executing #13. Additional rules, for example A processor may not perform the SLEEP operation while in mastermode, may be imposed to ensure that the system of processors does not become deadlocked. When a processor has finished mastermode processing and is ready to return to normalmode, it does so in practice by calling a procedure named NORMALMODE, which consists of the EXIT block.

#14 - Store the address of this processor's wakeup word in the right half of AC and clear the left half to zero.

#15 - Form the bit-by-bit "exclusive OR" of the contents of AC and the shared flag word, and store the result in both AC and the shared flag word. This must be done by a single machine instruction. Note that the left half of the shared flag word is not changed in this step, since the left half of AC contained zero. Also note that the right halves of AC and the shared flag word will now contain zero if and only if the right half of the shared flag word contained the address of this processor's wakeup word just before this step was done.

#16 - Test the value now in the right half of AC against zero.

#17 - Store the value now in the left half of the shared flag word in the right half of that word, leaving the left half unchanged. This must be done by a single machine instruction. (Note: The value stored might not be the one which was in the left half of the shared flag word at the time #15 was done, since another processor may have done #5 in the meantime.)

#18 - Store this processor's number in AC.

#19 - Interchange the contents of AC and the contents of this processor's wakeup word.

#20 - Test the contents of AC against zero.

#21 - Perform the SLEEP operation. Do not continue beyond this point until some other processor performs the WAKEUP(K) operation, where K is this processor's number.

#22 - Move the number now contained in this processor's wakeup word into AC.

#23 - Perform the WAKEUP(M) operation, where M is the number now contained in AC.

#24 - No processing is done at this step. It represents the point at which the processor is ready to return to normalmode. In practice, this step is the return point of the NORMALMODE procedure.

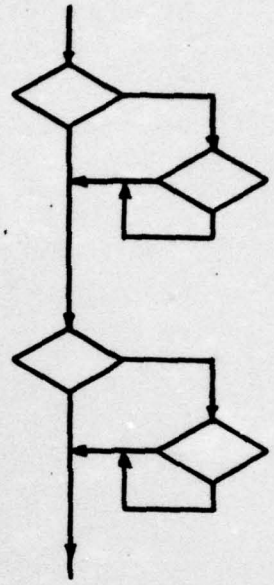
6.6 INITIAL STATE OF THE SYSTEM.

It is assumed that when each processor is first started, it begins at the START point of the flowchart and follows the cyclic program specified therein. The only assumption about the speed of a given processor is that it continues to progress at some rate unless it has delayed itself indefinitely by performing the SLEEP operation or has halted while in normalmode. An aspect of the algorithm not shown in the flowchart is the required initial state of the system prior to the starting of any processor. The requirements are that the shared flag word contain zero and that there be no pending wakeups in effect for any processor.

6.7 EXPLANATION OF SOLUTION ALGORITHM.

We described above each step of the flowchart, but have still not explained how or why the algorithm works. To do this, it will help to consider the various interactions that can occur between processors. Recall that the activity of a processor from the time it first requests to enter

mastermode until it is ready to return to normalmode is called a pass through mastermode. The processing done from step #3 to step #24 of the flowchart represents one pass. This part of the flowchart is shown on the right in "skeleton" form. Here only the decision points at which the processor can take alternate execution paths are shown. Clearly there are three ways to traverse the upper part of the flowchart and three ways to traverse the lower part, and hence there are nine different paths a processor might take on a given pass. Previously nine kinds of pass were defined based on the interaction between a processor and its leader and its follower. It is not a mere coincidence that there are nine kinds of pass and also nine flowchart paths. In fact, there is an exact correspondence between which of the nine paths a processor is following and what type of pass it is making. Point A and point B, used in defining the pass types, correspond to step #5 and step #15 of the flowchart, respectively.



Now consider the path followed by a processor making a type N pass. That the pass is type N indicates that the processor has neither a leader nor a follower, which means that (1) there were no processors between #5 and #15 when the given processor did #5, and (2) the processor will do #15 before any other processor does #5. It will be shown

later that the right half of the shared flag word contains zero whenever there are no processors between #5 and #15. Thus the processor making a type N pass, in doing #5, sets the right half of AC to zero. As a result, it takes the "yes" path from #6 and enters mastermode without delay. Later, when the processor is ready to do #15, the right half of the shared flag word still contains the address of the processor's wakeup word, which it put there when it did #5. Thus, in doing #15, the processor sets the right half of AC to zero. As a result it takes the "yes" path from 16 and returns to normalmode at once. Note that as soon as there is no longer a processor between #5 and #15 (i.e., as soon as the processor does #15), the right half of the shared flag word again contains zero, as required. Thus if another processor now makes a type N pass, it will work exactly like the one just described.

Next we will determine the path followed by a processor, say Processor Q, which is making a type L pass (either type L1 or type L2). Since the pass is type L, Processor Q has a leader, which we denote as Processor P. From the definition of a type L pass, we know that at the time Processor Q does #5, one or more processors are between #5 and #15, and the one which did #5 most recently is Processor P. We will show later that whenever there are one or more processors between #5 and #15, the right half of the shared flag word is nonzero and the left half of the shared flag word contains the address of the wakeup word of the

processor which most recently did #5. Thus, in doing #5, Processor Q stores a nonzero value in the right half of AC and the address of Processor P's wakeup word in the left half of AC. (Furthermore, it stores its own wakeup word's address in both halves of the shared flag word, so the condition stated above still holds.) As a result, it takes the "no" path from #6.

Let us now look ahead to the time when Processor P has finished its mastermode processing and is ready to return to normalmode. (Processor P might be asleep waiting for its turn in mastermode when Processor Q does #5, but in that case it will eventually be waked up by its own leader in the same way that it in turn wakes up Processor Q, which is the case we will describe.) Just before Processor P does #15, the right half of the shared flag word will contain the address of the wakeup word not of Processor P, but rather of Processor Q, so the "exclusive or" operation will place a nonzero result in the right halves of both AC and the shared flag word. (No other processor has done #5 in the meantime, since such a processor would become Q's follower, but Q is making a type L pass and hence has no follower.) As a result of the nonzero value in the right half of AC, Processor P will take the "no" path from #16.

We have just seen that because Processor Q is making a type L pass, it will take the "no" path from #6, and its leader, Processor P, will (eventually) take the "no" path

from #16. We now have to consider whether Processor Q reaches #8 before or after Processor P reaches #19, because this determines whether Processor Q is making a pass of type L1 or type L2. In the normal case, corresponding to a type L1 pass, Processor Q does #8 before Processor P does #19. Consider what happens when Q does #8. Before, in step #7, Q stored in J the address of P's wakeup word, which it got from the left half of the shared flag word back at step #5. At #7 Processor Q also stored its own number in AC. Thus at #8, Q stores its own number in P's wakeup word and stores in AC the value previously in P's wakeup word. But this value is zero, because Processor P cleared its wakeup word when it did #3, and no other reference to P's wakeup word is made until P does #19. Thus Q will take the "yes" path from #9 and will perform the SLEEP operation at #11. Here Processor Q's progress comes to a halt for the time being.

In the meanwhile, Processor P has been working its way toward #19. At the time Q did #8, P was most likely doing #13, but it could in fact have been just getting ready to do #6, or have been asleep itself at #11, or have already reached #18. (That is, we only know with certainty that P has already done #5 and has not yet done #19 when Q does #8 in the case under discussion.) At any rate, we know that P will eventually take (or has already taken) the "no" path from #16, for reasons already given. At #17 Processor P restores the shared flag word to its condition before P did #15 by copying its left half (which contains the address of

Processor Q's wakeup word, inasmuch as Q has no follower) into its right half. This is necessary so that the test at #16 will work correctly when Processor Q does it later. At #19 P stores its number in its own wakeup word, and stores in AC the value previously in its wakeup word. This value is Processor Q's number, put there by Q at step #8. Since this value is nonzero, P takes the "no" path from #20. At #23 it wakes up the processor whose number is in AC, namely Processor Q, and then returns to normalmode to repeat its cycle. As soon as P does #23, Q is allowed to proceed past #11. Its subsequent activity is like that of a processor making a type N pass, as already described.

In the case just discussed a processor making a type L1 pass (Processor Q) went to sleep prior to entering mastermode and was later wakened by its leader (Processor P). The sequence was distinguished as type L1 by the fact that Q did #8 before P did #19. Even in this sequence, it is possible that P will do #23, performing the WAKEUP(Q) operation, before Q reaches #11 and does the SLEEP operation. If this happens, a pending wakeup goes into effect for Processor Q, which will then wake up immediately upon performing the SLEEP operation. Thus the processors still behave as desired. This result reflects a required characteristic of the SLEEP/WAKEUP operations, namely that a WAKEUP directed at a processor which is not sleeping causes the next SLEEP by that processor to have no effect other than to cancel the pending wakeup.

Next we will consider the "abnormal" case, corresponding to a type L2 pass, in which Processor P does #19 before Processor Q does #8. In this case Processor P's wakeup word still contains zero just before P does #19, so in doing #19 P sets AC to zero. As a result, P takes the "yes" path from #20 and performs the SLEEP operation at #21. When Processor Q does #8, Processor P's number is moved from P's wakeup word (where it was put by P at #19) into the AC of Processor Q. Thus Q takes the "no" path from #9, wakes up at #10 the processor (Processor P) whose number is in AC, and then goes to sleep at #11. Processor P proceeds from #21 after Q does #10. At #22, Processor Q's number is moved from P's wakeup word (where it was put by Q at #8) to the AC of Processor P. Thus at #23 Processor P performs the WAKEUP(Q) operation, after which it returns to normalmode for another cycle. After P does #23, Q proceeds from #11, enters mastermode, and completes its pass as if it were type N.

We have now fully described the interaction which takes place between a processor making a type L pass and its leader. When a processor makes a type F pass, a similar interaction occurs between the processor and its follower. Suppose Processor P is making a type F pass (either type F1 or type F2) and its follower is Processor Q. In describing this case we will concentrate mainly on the differences between it and the previous case, in which Processor Q was

making a type L pass. (Note that now Q is making either a type L or type LF pass.)

Since Processor P is making a type F pass and hence has no leader, we know that it will take the "yes" path from #6 and will be the only processor between #5 and #15 when Processor Q does #5. This is in contrast to the previous case, in which P might or might not have had its own leader and in which there might have been any number of processors between #5 and #15 when Q did #5. But whether or not P has a leader, Q still finds a nonzero value in the right half of the shared flag word and finds the address of P's wakeup word in the left half of the shared flag word when it does #5, so Q's behavior in the ENTRY block is the same as before. In particular, Q takes the "no" path from #6 after leaving the address of its own wakeup word in both halves of the shared flag word at #5.

Now we return to Processor P's activity after Processor Q has done #5. Since Q might have a follower, one or more other processors might do #5 before P does #15. Even so, the right half of the shared flag word will contain a wakeup word address other than P's own, and therefore in doing #15 P will store a nonzero value in the right half of AC and will subsequently take the "no" path from #16. When P does #17, it may or may not restore the shared flag word to its condition just before P did #15, because some processor might have done #5 while P was between #15 and #17. But in

this case P can do no harm when doing #17, since both halves of the shared flag word will already contain the same number. (The case previously discussed, in which Q has no follower, is the case which makes #17 necessary.)

We have just seen that a processor making a type F pass will take the "no" path from #16 and its follower will take the "no" path from #6. (We noted a similar result before from the viewpoint of the follower.) The interaction which now occurs between leader and follower is identical to that of the previous case. If Q does #8 before P does #19 (in which case P is making a type F1 pass), then Q goes to sleep after taking the "yes" path from #9 and is later awakened by P, which has taken the "no" path from #20. On the other hand if P does #19 before Q does #8, then P (which is now making a type F2 pass) will go to sleep at #21 and will be awakened later by Q at #10, after which it will return the favor by performing a WAKEUP(Q), allowing Q to get past #11. (Of course in each case it is possible for the WAKEUP to precede the SLEEP, causing the SLEEP in effect to be ignored.)

All the interactions that can occur between a processor and its leader or its follower have now been described. Although the case of a processor making a type LF pass has not been covered specifically, it may be described by combining the previous cases. That is, if a processor is making a type LF pass, its activity in the ENTRY block is

the same as that of a processor making a type L pass, and its activity in the EXIT block is the same as that of a processor making a type F pass. This activity may be summarized as follows:

(1) A processor takes the "no" path from #6 if and only if it has a leader.

(2) A processor takes the "no" path from #16 if and only if it has a follower.

(3) A processor takes the "yes" path from #9 if and only if it does #8 before its leader does #19.

(4) A processor takes the "yes" path from #20 if and only if it does #19 before its follower does #8.

These four statements emphasize the significance of the leader/follower relationship among the interacting processors, and would be highly useful in proving the correctness of the algorithm. Unfortunately we have up to now merely asserted, not proved, that these statements are true. As we begin to construct a rigorous proof, we must be careful not to use arguments which depend explicitly or implicitly on the validity of these statements.

6.8 SATISFACTION OF PROBLEM REQUIREMENTS.

Before going on to the correctness proof of the next chapter, let us note that several of the requirements

appearing in the statement of the mastermode/normalmode problem are clearly satisfied by the proposed algorithm. For example the solution requires N+1 shared memory locations, meeting the upper limit of 2N locations. Furthermore these locations do not have to be contiguous. The total number of processors, N, does not appear in the data base of the solution, so there is no predetermined limit on this number, and dynamic allocation of shared storage is feasible. Finally, note that the sections of the flowchart representing the ENTRY and EXIT blocks are loop-free even at the primitive operation (machine instruction) level. Thus the algorithm employs no busy waiting, cyclic sleeping, or iterated testing, all of which would involve some kind of loop. It remains to be established that the algorithm satisfies the following requirements:

(1) At most one processor can be in mastermode at a given time.

(2) Processors enter mastermode in the same order that they pass point A (i.e., execute step #5).

(3) Any processor which passes point A eventually completes its pass through mastermode and returns to normalmode (i.e., deadlock does not occur).

A rigorous proof that the algorithm does indeed satisfy these conditions is given in the next chapter. The discussion given up to now has merely provided an informal "explanation" of the algorithm, and at this point the reader should be reasonably familiar with the operation of the algorithm and the interactions that occur among processors competing to enter mastermode.

6.9 EFFICIENCY OF THE SOLUTION ALGORITHM.

As we noted earlier, the algorithm presented above meets all the conditions imposed in the statement of the mastermode/normalmode problem. Several of these conditions were included in the hope that they would make the solution highly efficient. In other words, it was hoped that a solution which met all of the conditions would be one in which the processors would incur a relatively low "overhead" in their use of the mastermode/normalmode facility. We are now in a position to determine whether this hope has been realized.

When using the approach to process synchronization described in Chapter 4, we normally expect processors to spend most of their time in normalmode, only occasionally entering mastermode for relatively brief intervals. Under such conditions, the most common kind of pass for a processor to make is the type N pass, in which the processor does not interact with other processors either when entering

or leaving mastermode. To consider a given solution efficient, we would certainly require low overhead for a processor making a type N pass through mastermode. (We will consider all processing done in the ENTRY and EXIT blocks to be overhead, since a processor making a type N pass would have performed correctly if it had simply begun its mastermode processing and returned to normalmode without executing the steps of the algorithm in the ENTRY and EXIT blocks.)

During a type N pass a processor takes the "yes" path from both #6 and #16. After such a processor calls the MASTERMODE procedure (i.e., reaches the ENTRY block in the flowchart), the sixth machine instruction executed is a return to the calling program, at which point the processor has successfully entered mastermode. After the processor calls the NORMALMODE procedure (i.e., reaches the EXIT block), the fourth machine instruction executed is a return to the calling program, at which point the processor has successfully returned to normalmode. (These instruction counts are based on the fact that steps #3, #5, #6, #12, #14, #15, #16, and #24 each require one machine instruction and step #4 requires two machine instructions. An assembly language program which implements the algorithm is listed in Appendix B.)

In addition to the above processing, one machine instruction is required for each of the two procedure calls (to the MASTERMODE and NORMALMODE procedures). Thus a processor making a type N pass through mastermode incurs an overhead of 12 machine instructions. By comparison, using the "bakery algorithm" described in Chapter 5 the overhead in a corresponding situation is 7N+18 machine instructions, where N is the total number of processors. (This figure was obtained from an efficient assembly language implementation of the bakery algorithm as given in Appendix A.) Thus for example if there are ten processors using the mastermode/normalmode facility, a processor which entered and exited mastermode while all other processors remained in normalmode would incur an overhead of 88 machine instructions using the bakery algorithm. On the basis of this comparison it seems fair to consider the algorithm presented in this chapter a highly efficient one and to conclude that the conditions imposed in the statement of the mastermode/normalmode problem have been justified by the quality of the resulting solution.

CHAPTER 7

PROOF OF CORRECTNESS OF THE SOLUTION

7.1 INTRODUCTION

In this chapter we will develop a rigorous proof that the solution to the mastermode/normalmode problem presented in the last chapter is correct. As noted on page 6-27, we have to establish that processors are granted mutually exclusive first-come-first-served access to mastermode and that any processor which requests to enter mastermode eventually does so and finally returns to normalmode. The proof is complicated, and we will have to take great care to avoid the fallacy of basing certain proofs on implicit assumptions which in reality depend on later results. If the reader wishes to follow the proof in detail, he is advised to refer continually to the flowchart which folds out from page 7-57.

7.2 AN IMPORTANT ASSUMPTION.

A rigorous proof of correctness might be impossible (instead of merely being very difficult) without the

following assumption: The SLEEP and WAKEUP operations only occur where shown explicitly in the flowchart. In particular, no processor performs SLEEP or WAKEUP operations during normalmode processing at step #2 or mastermode processing at step #13 of the algorithm. We will depend on this assumption when performing the correctness proof, but will have to violate it when actually using the MASTERMODE and NORMALMODE operations. To understand this violation of the assumption, recall that the general procedure for solving resource allocation problems described in Chapter 4 required the use of SLEEP and WAKEUP operations for efficient control of processor delay. That is, the same SLEEP/WAKEUP mechanism which is used to implement the MASTERMODE and NORMALMODE operations, as shown in the flowchart, is also used at a higher level to control the activity of processors competing for an arbitrary shared resource, and such use requires SLEEP and WAKEUP operations to occur during normalmode and mastermode processing at steps #2 and #13. (The latter activity is said to occur at a "higher level" because the Chapter 4 procedure for resource allocation treats MASTERMODE and NORMALMODE as primitive operations, with no cognizance of the fact that performing either operation may cause one or more SLEEP or WAKEUP operations to occur.)

Due to the rudimentary nature of the SLEEP/WAKEUP mechanism, a processor cannot distinguish the wakeups which it receives in connection with its use of the

MASTERMODE/NORMALMODE operations from those it receives in connection with its (higher-level) use of a shared resource. Hence the possibility exists for improper system behavior due to unexpected interactions between these two uses of the SLEEP/WAKEUP mechanism. To rule out this possibility, after proving the correctness of the algorithm under the assumption that SLEEP and WAKEUP only occur where shown in the flowchart, we must establish that additional SLEEP and WAKEUP operations performed according to the pattern programs on pages 4-7 and 4-9 will not lead to improper system behavior. We will consider this issue again in Section 7.11 after completing the correctness proof.

7.3 OUTLINE OF THE PROOF PROCEDURE.

The correctness proof to be given in this chapter is long and complex, providing many chances for the reader to become lost in details and lose track of the overall objective. For that reason we will begin by listing the major steps to be performed in the course of the proof. The reader is encouraged to refer back to this outline, which will hopefully restore his perspective of the overall proof procedure, whenever he begins to feel overwhelmed by the massive detail and intricate reasoning required to complete the proof. The main steps of the proof are as follows:

(1) Determine the values which will appear in the left and right halves of the shared flag word and the circumstances under which these values can be altered. (Lemmas 1 through 5.)

(2) Develop a rigorous definition for the terms "leader" and "follower" which were used informally in the preceding chapter. (Definitions 1 and 2.)

(3) Define the concept of a "chain" of processors, which is a group of processors that enter mastermode one after the other, and describe a way of representing processor activity by drawing a "chain diagram". (Definition 3.)

(4) Establish a correlation between the leader/follower relationship and certain patterns appearing in the chain diagram. (Proposition 1.)

(5) Devise a definition which allows the "activity associated with a given chain" to be correlated with the execution of specific flowchart steps by specific processors. (Definition 4.)

(6) Recognize and rigorously define a "rest state" which the system of processors occupies when no processor interactions related to mastermode access are in progress. (Definition 5.)

(7) Show that once the system reaches the rest state, the processors will never again be influenced by activity associated with any previous chains. (Proposition 2.)

(8) Show that if the system is in the rest state just before the start of a given chain, the first processor of that chain will enter mastermode without delay. (Proposition 3.)

(9) Determine the values which will appear in the shared wakeup words associated with the processors of a particular chain, given that the system was in the rest state just before the start of that chain. (Proposition 4.)

(10) Show that if the system is in the rest state just before the start of a given chain, then all the processors of that chain which have followers (i.e., all but the last processor of the chain) are granted mutually exclusive, deadlock-free, first-come-first-served access to mastermode. (Proposition 5.)

(11) Show that if the system is in the rest state just before the start of a given chain, then the results of Proposition 5 also apply to the last processor of the chain, provided the chain has a last processor. (Proposition 6.)

(12) Show that if the system is in the rest state just before the start of a chain which terminates (i.e., has a last processor), then the system will again be in the rest state at some time after the end of that chain and before

the start of the next chain. This result allows us to show by induction that the system is in the rest state just before the start of every chain, and hence that Propositions 3 through 6 apply to every chain. (Proposition 7.)

(13) Show that although the proofs developed for the above results depend on the assumption given in Section 7.2 that SLEEP and WAKEUP operations only occur where shown explicitly in the flowchart, the conclusions remain valid when additional SLEEP and WAKEUP operations occur in accordance with the general procedure for resource sharing developed in Chapter 4. (Proposition 8.)

7.4 THE CONTENTS OF THE SHARED FLAG WORD.

Before beginning a detailed proof of correctness of the algorithm, we will derive some important preliminary results concerning the values appearing in the left and right halves of the shared flag word. We will denote these halfwords by (FLAG)L and (FLAG)R, in agreement with the notation used in the flowchart. Our results are expressed in Lemmas 1 through 5.

Our first objective is to determine which flowchart steps can alter the contents of the shared flag word. (As noted in Chapter 6, we assume that the shared locations used in the algorithm are altered only as shown in the flowchart.) The steps which affect the shared flag word

clearly include #5, #15, and #17. To these we must tentatively add #8, because at step #8 an exchange occurs between a processor's AC and the location whose address is in that processor's J register, and as far as we know J might contain the address of the shared flag word. Now the value in J when #8 is done was moved there at #7 from the left half of AC, which in turn was loaded from (FLAG)L at the time #5 was done. Only if this value is the address of the shared flag word can #8 affect the shared flag word. But can (FLAG)L ever contain the address of the shared flag word? Of course it contains zero (which might be the address of the shared flag word) initially, before any processor has done #5, but in this case (FLAG)R also contains zero, so the first processor to do #5 will take the "yes" path from #6, bypassing #8. When #5 is done for the first time, and every time thereafter, the address of some processor's wakeup word is stored in (FLAG)L. Thus if (FLAG)L is ever to be set to the address of the shared flag word, it can only happen at #8. But this means that the execution of #8 can alter the shared flag word only if the shared flag word has already been altered by the execution of #8, which rules out the possibility of such an alteration ever happening for the first time. This result is summed up in the following lemma.

Lemma 1. Whenever the contents of the shared flag word are altered, it is the result of some processor doing either #5, #15, or #17.

Since (FLAG)L is not affected by #15 or #17, we see from Lemma 1 that it only changes when a processor does #5. This leads immediately to the following result:

Lemma 2. After #5 has been done for the first time by any processor, (FLAG)L always contains the address of the wakeup word of the processor which did #5 most recently.

Now we are ready to consider the contents of (FLAG)R. In particular we want to know when the value in (FLAG)R can be zero, because this value (after being moved to the right half of AC) is compared against zero in the critical tests at #6 and #16. That is, the path taken from #6 by a given processor reflects the condition (zero or nonzero) of (FLAG)R just before #5 was done by that processor, and the path taken from #16 reflects the condition of (FLAG)R just after #15 was done. Now at step #5 (FLAG)R is set to the (nonzero) address of a processor's wakeup word, and at step #17 it is set to (FLAG)L, which by Lemma 2 also contains the address of a processor's wakeup word. Thus #15 is the only step whose execution could set (FLAG)R to zero. Clearly, if #15 is done by the processor which most recently executed step #5, without an intervening execution of #15 by a different processor, then (FLAG)R will be set to zero by the "exclusive or" operation at #15. (Note that an intervening execution of #17 does not change this result, since any processor doing #17 sets (FLAG)R to the address of the

wakeup word of the processor which most recently did #5, as is clear from Lemma 2.)

But there is another way (FLAG)R might become zero, as the following example shows. Suppose that Processors 1, 2, and 3 have their wakeup words at the shared locations whose addresses are 1, 2, and 3 respectively, and suppose Processor 3 has done #5, so that (FLAG)L and (FLAG)R each contain the number 3. If Processor 2 does #15 now, it takes the "exclusive or" of its AC (whose right half contains 2) and the shared flag word (whose left and right halves contain 3) giving a result whose left half is 3 and whose right half is 1. This result is stored in the shared flag word, so the number in (FLAG)R is changed from 3 to 1. Now suppose Processor 1 does #15 before any processor does #5 or #17. Since (FLAG)R contains 1, Processor 1 sets (FLAG)R to zero in doing #15. Thus (FLAG)R has become zero even though #15 has not been done by the processor which most recently did #5 (namely, Processor 3). Clearly this would not have happened if some processor had done #5 or #17 between the times #15 was done by Processors 2 and 1, because a processor doing #5 would set (FLAG)R to the address of its own wakeup word, and a processor doing #17 would set (FLAG)R back to 3 (which is the address of the wakeup word of the processor which most recently did #5). The above argument may be summarized as follows:

Lemma 3. The contents of (FLAG)R can only be changed from a nonzero to a zero value as a result of one of the following:

(1) #15 is done by the processor which did #5 most recently.

(2) #15 is done two or more times in succession by different processors, without an intervening execution of #5 or #17 by any processor.

Note that the lemma does not say that (FLAG)R must become zero under the stated conditions, but rather that it is possible for it to become zero only under these conditions.

We will prove later that the second condition of Lemma 3 cannot occur. In order to keep our results on the shared flag word together, that result is given here without proof.

Lemma 4. If a processor does #15, and sometime thereafter the same or another processor does #15 again, some processor will have executed #5 or #17 in the meantime.

Now suppose a processor has just done #15 for the first time since #5 was most recently done. If the processor is not the one which did #5, it will not have set (FLAG)R to zero in doing #15. Furthermore, before #15 is done again by any processor, Lemma 4 says that #5 or #17 will be done by some processor. But as soon as any processor does #5 or

#17, it is again true that (FLAG)R contains the address of the wakeup word of the processor which most recently did #5, so only this processor can set (FLAG)R to zero by doing #15. This leads to the following result:

Lemma 5. The contents of (FLAG)R are changed from a nonzero to a zero value when and only when #15 is done by the processor which did #5 most recently.

Of course we must remember not to use Lemma 4 or Lemma 5 until we have proved Lemma 4.

7.5 RIGOROUS DEFINITION OF LEADER AND FOLLOWER.

In the preceding chapter we used the terms "leader" and "follower" to simplify the description of processor interactions. However, we did not define the terms very carefully at the time. In particular, we disregarded the fact that each processor is engaged in a cyclic process and may make many passes through mastermode. Because of this fact, a given processor, say Q, may have one processor as its leader on one pass and a different processor as its leader on a later pass. In such a case, is the first leader of Q still considered Q's leader on the later pass, giving Q two leaders? If not, exactly when did Q lose its first leader and gain its second one? The following definitions allow us to answer such questions precisely.

Definition 1. "Processor P is the leader of Processor Q" means that the following statements are true:

(1) Q is between #5 and #15.

(2) P was between #5 and #15 at Tq, which denotes the time at which Q did #5 most recently.

(3) No processor did #5 after Tpq and before Tq, where Tpq denotes the most recent time prior to Tq at which P did #5.

Definition 2. "Processor Q is a follower of Processor P" means that P is the leader of Q.

It is important to realize that these definitions refer to conditions which may or may not exist at a given time. That is, the answer to the question "Is P the leader of Q?" may be "yes" at one time, "no" at a later time, and "yes" again at a still later time. Whenever a statement involving leaders and followers is made, it should be interpreted as if a phrase such as "at the present time" or "at the time implied by the context" were added to the statement.

The following consequences of Definitions 1 and 2 are noted in passing. They may give the reader a better idea of what it means for a processor to be a leader or a follower.

(1) When a processor becomes a follower, it does so at #5. Thereafter it continues to be a follower until and only until it does #15.

(2) When a processor becomes a leader, it does so at some point between #5 and #15. Thereafter it continues to be the leader of a particular follower until that follower does #15.

(3) It is possible for a processor to have more than one follower at a given time. For instance, it could have one follower associated with its current pass and one associated with its previous pass. (After establishing the correctness of the algorithm, we can show that a processor cannot have more than two followers at a given time.)

(4) It is not possible for a processor to have more than one leader at a given time.

(5) It is possible for a processor to be a leader and a follower at the same time. In fact it could be the leader and the follower of the same processor at a given time.

(6) It is not possible for a processor to be its own leader or follower.

7.6 REPRESENTATION OF PROCESSOR ACTIVITY.

The following procedure gives a way of recording the activity which occurs when several processors concurrently perform the flowchart algorithm:

Starting at a time before any processor has done #5, record each execution of #5 by writing down the number of the executing processor. Write these processor numbers from left to right in the order that the processors do #5. As each execution of #5 is recorded, note whether the processor which did #5 just before (i.e., the processor whose number is to the left of the number just written) is still between #5 and #15. If so, place a dash between the last two numbers written.

Definition 3. The string of numbers and dashes produced by this procedure will be called a "chain diagram", and each group of processors joined by dashes will be called a "chain". The leftmost processor in each chain is designated the "first processor" of the chain. The rightmost processor, if it exists, is designated the "last processor" of the chain. The "beginning of a chain" denotes the time at which the first processor of the chain executes #5, and the "end of a chain" denotes the time at which the last processor of the chain executes #15. The system of processors is said to be "between chains" during the time interval after the end of one chain and before the beginning of another.

Below is shown a chain diagram which might result when N=5 (i.e., there are five processors numbered 1 to 5).

2 2-4-1 3-5-3-2 4-2-5-1-2-3-4-1-

The final dash means that observation of the processors stopped here, and the last chain may or may not extend beyond this point. In the third chain above, Processor 3 is the first processor and Processor 2 is the last processor. This chain ends when Processor 2 does #15. (The definition does not imply that the other processors of the chain will have done #15 by this time.) In the given diagram, it is not possible to determine whether the fourth chain has a last processor.

The procedure for making a chain diagram requires that the first number be recorded when #5 is first done by any processor. We do not rule out the possibility, however, that all the processors may be started over again (at the START point of the flowchart) at some later time. Before a new startup of the processors, the shared flag word must be reset to zero and all pending wakeups (if any) must be cancelled. A new chain diagram results from each startup of the processors. The following proposition shows the significance of the chain diagram.

Proposition 1. When a chain diagram is created as described above, the sequence P-Q will appear in the diagram if and only if Processor Q became the follower of Processor P upon doing #5.

Proof. This proposition follows immediately from Definition 1. In fact, the procedure for making the chain diagram was developed with Definition 1 in mind, so that the

chain diagram would have the property given in this proposition. □

Proposition 1 shows that all leader/follower relationships appear in the chain diagram. For example, consider the fourth chain of the diagram shown above. The first dash in this chain signifies that Processor 2 became a follower of Processor 4 at the indicated time (the indicated time being the fourth execution of step #5 by Processor 2 since the processors were started). Then when Processor 5 did #5, it became Processor 2's follower. Processor 2 had to be between #5 and #15 at this time, and hence was still a follower of Processor 4. Thus there was some time period when Processor 2 was both a follower of Processor 4 and the leader of Processor 5. Similarly, in the third chain it is seen that there was a time period when Processor 5 was both the leader and the follower of Processor 3. When Processor 2 did step #5 in the third chain it became a follower of Processor 3, but the diagram does not indicate whether Processor 5 was still Processor 3's follower at that time. This depends on whether Processor 5 did #15 before or after Processor 2 did #5. In the latter case, Processor 3 would have had two followers at the same time.

7.7 INTERFERENCE BETWEEN CHAINS.

In the last chapter we introduced the idea of a "pass through mastermode" corresponding to the execution of steps #3 through #24 by a given processor. It is clear that a processor does #5 one time, and hence appears once in the chain diagram, for each pass it makes. Consequently we can associate all actions of a processor that occur between steps #3 and #24 inclusive with a particular appearance by that processor in the chain diagram. Likewise we can identify all the activity associated with a particular chain of the chain diagram. The following definition makes that identification explicit.

Definition 4. The "activity associated with a given chain" denotes collectively the execution of steps #3 through #24 of the algorithm by all processors during the pass or passes on which they appear in the given chain.

Note that by this definition, the activity associated with a given chain starts before the beginning of the chain and continues after the end of the chain. For example, consider the chain diagram on page 7-14. After Processor 4 completed its first pass (on which it appeared in the second chain) it may have done #3, starting its second pass, before the beginning of the third chain. It would then have had to remain between #3 and #5 until after the end of the third chain, since we see from the diagram that the fourth chain began when Processor 4 did #5 on its second pass. In the

case just described, some activity associated with the fourth chain occurred before the beginning of the third chain. It is also easy to think of examples in which activity associated with the second chain would continue to occur after the start of the third chain.

The fact that activity associated with earlier and later chains can occur between the beginning and end of a given chain (which we refer to as the "present" chain) raises a significant question: Can such activity interfere with the proper interaction of processors appearing in the present chain? The answer to this question is intimately related to a certain state in which the entire system of processors can be, which we will call the "rest state". The rest state plays an important part in the proof of correctness and is analogous to the "homing position" of Dijkstra and Habermann (Ha67, Di68a). We define the rest state as follows:

Definition 5. The system of processors is said to be in the "rest state" when the following conditions hold:

(1) If any processor is between #3 and #5, its wakeup word contains zero.

(2) The right half of the shared flag word contains zero.

(3) There are no processors between #5 and #15 or between #16 and #23.

(4) Any processor between #15 and #16 will take the "yes" path from #16.

(5) No pending wakeups are in effect.

Note that the system of processors is in the rest state initially, before any processor has done #5. Since in the rest state there can be no processors between #5 and #15, it is clear that the system must be between chains when in the rest state. Another significant property of the rest state is expressed in the following proposition.

Proposition 2. Once the system reaches the rest state, no shared storage location will ever again be affected by any activity associated with a previous chain, nor will any processor perform the WAKEUP operation as a result of any such activity.

Proof. When the system is in the rest state, any processor in a previous chain must have already passed #15, and either will take (or has taken) the "yes" path from #16, or will already be past #23 if it took the "no" path from #16. Therefore such a processor cannot alter the shared flag word (see Lemma 1) unless it appears in a new chain by doing #5 again, nor can it do #8 or #19, the only steps which can alter the shared wakeup words. By similar reasoning no processor from a previous chain can do #10 or

#23 once the system has reached the rest state, and these are the only steps in which the WAKEUP operation is performed. This establishes Proposition 2. □

Proposition 2 identifies a sufficient condition for ruling out any possibility that activity associated with earlier chains might interfere with the present chain. The sufficient condition is that the system of processors be in the rest state at some time between the end of the previous chain and the beginning of the present chain. As we noted above, the system is clearly between chains when in the rest state. However the converse question (namely, is the system in the rest state when between chains) is much more difficult to answer and in fact will not be resolved until we have proved Proposition 7. Until that time it will be necessary to include, among the given conditions for each proposition that we prove, a requirement that the system be in the rest state prior to the beginning of the chain to which the proposition applies.

7.8 RELATION OF CHAIN DIAGRAM TO FLOWCHART PATHS.

In Chapter 6 we described the activity of various processors in terms of the paths they followed in the flowchart while performing the algorithm. In the present chapter we have introduced a new way to describe such activity, namely the chain diagram. With the following

proposition we begin to show a correlation between these two approaches.

Proposition 3. Suppose the system is in the rest state at some time before the beginning of a given chain and after the end of any previous chain. Then the first processor of the given chain will take the "yes" path from #6.

Proof. When the system is in the rest state, (FLAG)R contains zero by Definition 5. But by Proposition 2, (FLAG)R will not be altered until a new chain begins. Thus the first processor of the new chain will set the right half of its AC to zero in doing #5, and hence will take the "yes" path from #6. □

Proposition 3 may be paraphrased as follows: Assuming that the system is in the rest state prior to the start of a given chain, the first processor of that chain will be allowed to enter mastermode without delay. We would certainly expect this to be the case, since we know that there can be no processors between #5 and #15 when the first processor of a chain does #5 under the given condition.

Recall that there is a shared storage location called a wakeup word associated with each processor. We will refer to the processor corresponding to a given wakeup word as the "owner" of that word. The following proposition describes the contents of the wakeup words.

Proposition 4. Suppose the system is in the rest state at a time, say time T1, before the beginning of a given chain and after the end of any previous chain. Then the following statements are true of the wakeup word of each processor in the given chain:

(1) It is set to zero when its owner does #3.

(2) It is set to its owner's number when and if its owner does #19 during the new chain.

(3) It is set to the number of a follower of its owner when and if any such follower does #8 during the new chain.

(4) It is not altered under any circumstances other than the ones just listed, between time T1 and the end of the new chain.

(5) It contains zero when its owner does #5 for the first time in the new chain.

Proof. Statements (1) and (2) are obvious from the flowchart. Considering Lemma 2 and the flowchart, it is clear that any processor doing #8 references the wakeup word of the processor which did #5 just before it did, which by definition is its leader if it has a leader. But it must have a leader, because otherwise it would be the first processor in the chain and by Proposition 3 would not be doing #8. Thus whenever a processor does #8 it references the wakeup word of its leader, setting that word to its own

number. This proves statement (3). To prove (4), note from Proposition 2 that after time T1 no processor from a previous chain can alter a wakeup word. Thus only processors of the new chain can affect a wakeup word between time T1 and the end of the chain. From the flowchart the only steps which can alter a wakeup word are #3, #8 and #19, and the effects of each of these have been accounted for in statements (1), (2), and (3). Thus statement (4) is established. To prove (5), let P be any processor of the new chain. It is clear from the statements already proved that P's wakeup word cannot change from zero to a nonzero value between time T1 and the time P first does #5. Therefore if P does #3, setting its wakeup word to zero, after time T1, then the wakeup word must still contain zero when P does #5. On the other hand, suppose P does #3 before time T1. In this case P will be between #3 and #5 at time T1, so by the first part of Definition 5 P's wakeup word must contain zero at time T1. Hence in this case also, P's wakeup word will still contain zero when P first does #5. This proves (5) and finishes the proof of the proposition.□

7.9 THE MAIN PROPOSITION.

At this point it may seem to the reader that our correctness proof is not getting anywhere, since we have not yet touched upon the primary issues, as listed on page 6-27, that we set out to prove. Fortunately that situation is

about to change. The proposition given in this section establishes that our proposed solution to the mastermode/normalmode problem has every property required in the problem statement. This proposition will not complete the proof of correctness, however, because it does not apply to all of the processors appearing in the chain diagram. After proving the proposition, our remaining task will be to extend its applicability to cover every appearance of every processor in the chain diagram.

Proposition 5. Suppose the system is in the rest state at some time before the beginning of a given chain and after the end of any previous chain. Then the following statements hold for a processor of the given chain during any pass on which that processor has a follower:

(1) Its follower will take the "no" path from #6.

(2) It will eventually do #19, and moreover will do so before its follower does #12.

(3) It will eventually do #24, but not before its follower does #8.

(4) Once it reaches #24, it has no wakeup pending and will not receive a later wakeup resulting from the pass it has just made.

(Note: With regard to the terminology used in part (4), we consider that a WAKEUP(P) operation "results from" a particular pass by P if the processor which performs the operation does so after finding P's number in a wakeup word where it was placed by P during the given pass.)

Proof. The proof will be by induction. We will first prove that the proposition holds for the first processor of the chain. Then we will prove that if the proposition holds for all processors in the chain up through a given one, it also holds for the follower of the given processor.

Let P be the first processor of the chain, and let Q be its follower. (If it has no follower, the proposition does not apply.) By Proposition 3, P takes the "yes" path from #6 and hence eventually reaches #15. We will now show by contradiction that no other processor of the current chain reaches #15 before P does. Suppose this is not the case, and let V be the first processor to do #15 in the current chain. Now when P did #5 it set FLAG(R) to a nonzero value, and it is clear from Lemma 3 that FLAG(R) cannot be set to zero at least until V does #15. Hence V finds a nonzero value in FLAG(R) at #5, takes the "no" path from #6, and eventually performs the SLEEP operation at #11. In order for V to proceed past #11 some processor of the current chain, say Processor W, must perform the WAKEUP(V) operation at #10, since Proposition 2 rules out a wakeup by a processor from a previous chain. Now for W to wake up V at

#10, it must have found V's number in its leader's wakeup word at #8. But until some processor passes #15, returns to normalmode, and begins another pass, no processor of the chain can have more than one follower. Hence when W does #8 it is making the first reference to its leader's wakeup word since a time at which the wakeup word contained zero, namely when the owner of the word did #5 (see part (5) of Proposition 4). Thus W must find zero, rather than V's number, in its leader's wakeup word at #8. This contradiction establishes that no processor of the current chain does #15 ahead of P, the first processor of the chain.

The above conclusion and Lemma 3 together imply that (FLAG)R is not set to zero when P does #15 (note that at least Processor Q must do #5 before P does #15, else Q would not be P's follower). Therefore P will take the "no" path from #16 and must eventually do #19. We also conclude that Q, having done #5 before P did #15, must take the "no" path from #6.

We now wish to show that Q cannot do #12 before P does #19. Since there can be no pending wakeup in effect for Q (by part (5) of Definition 5), it will suffice to prove that when Q goes to sleep at #11 it cannot be waked up except by P at #23. It is clear from the flowchart that before a processor can do the WAKEUP(Q) operation it must find Q's number in some wakeup word. This number must have been put there by Q itself, because one processor never puts

another's number into any wakeup word and because the wakeup word in question had to contain zero when its owner first did #5 in the current chain. But the only wakeup word referenced by Q so far in the current chain is that of its leader, P, at #8. Thus after Q goes to sleep at #11, another processor can wake Q up only after finding Q's number in P's wakeup word. Now by Proposition 4 the only processor which can do this is P itself. Therefore, a wakeup by P at #23 is the only possibility for the WAKEUP(Q) operation to be performed when Q is asleep at #11 (or has not yet reached #11). This makes it clear that P must do #19 before Q can do #12.

Next, we must show that P eventually does #24, but not before Q does #8. We have already established that P eventually does #19, and also that Q eventually does #8 (since Q takes the "no" path from #6). Two cases will now be considered. In Case I, we assume that Q does #8 before P does #19. In this case, P obviously does not do #24 before Q does #8, so we only have to show that P does in fact reach step #24. Now when Q does #8 it sets P's wakeup word to its own number, and that wakeup word must remain unaltered until P does #19 (see Proposition 4). Thus P will find a nonzero value in its wakeup word at #19 and take the "no" path from #20. This ensures that P will reach #24, since there is no possibility of blocking (i.e., the SLEEP operation is not performed) when the "no" path is taken from #20.

For Case II, we assume that P does #19 before Q does #8. By Proposition 4, P's wakeup word contained zero when P did #5 and cannot be altered thereafter until P does #19. Thus when P does #19 it sets its AC to zero and sets its wakeup word to its own number. As a result, P takes the "yes" path from #20 and performs the SLEEP operation at #21, beyond which point it cannot proceed until a WAKEUP(P) operation is performed by some processor of the present chain. We will now prove that a WAKEUP(P) can only be done by Q. Suppose some processor of the present chain other than Q, say Processor W, performs the WAKEUP(P) operation, and let V be the leader of W. (P cannot be the leader of W, because P is making its first appearance in the present chain and hence cannot have more than one follower before completing its current pass.) Now if W does the WAKEUP(P) operation at #10, it must have found P's number in V's wakeup word while doing #8. Or if it does the WAKEUP(P) operation at #23, it must have found P's number in its own wakeup word while doing #19 or #22. By Proposition 4, both of these wakeup words contained zero at the time their owners first did #5 in the present chain, and either word could later be set to P's number only if P were a follower of the owner of the word. But this is impossible, because P is the first processor in the chain and cannot become a follower before completing its first pass. Thus, by contradiction, only Q can perform the WAKEUP(P) operation during P's first pass.

Now we must prove that Q does in fact perform the WAKEUP(P) operation in the present case (Case II). Recall that P sets its wakeup word to its own number when doing #19, and Q has not yet done #8 at that time. By Proposition 4, P's wakeup word can be altered between the time P does #19 and the time Q does #8 only if P returns to normalmode, begins a new pass, and repeats #3 before Q does #8. But we have just proved that P cannot proceed past #21 until Q wakes P up. Thus when Q references P's wakeup word at #8, it still contains P's number, put there by P at #19. As a result Q will take the "no" path from #9 and perform the WAKEUP(P) operation at #10. Hence in Case II P will eventually do #24, but not before Q does #10, and thus surely not before Q does #8.

Finally we need to show that there is no pending wakeup in effect for P at the time P does #24, and that no processor performs the WAKEUP(P) operation after that time as a result of P's first appearance in the chain. Now in Case I, when P takes the "no" path from #20, Proposition 4 guarantees that P's wakeup word will remain unaltered from a time at which it contained zero (namely, when P did #5) until Q does #8. As a result, Q will find zero in P's wakeup word at #8, will take the "yes" path from #9, and will not do the WAKEUP operation at #10. In Case II, when P takes the "yes" path from #20, we proved above that Q does perform the WAKEUP(P) operation at #10. We also proved that no processor besides Q performs the WAKEUP(P) operation.

Thus the WAKEUP(P) operation is performed during P's first pass if and only if P has gone (or will go) to sleep at #21. As a result, P will not reach #24 until Q has already done the WAKEUP(P) operation, if it is going to do that operation. After P does #24, its wakeup word can be referenced again only by P itself at #19 (on a new pass) or by a new follower of P at #8. In either case P will already have repeated #3, clearing the number it placed in its wakeup word during its first pass. Hence any WAKEUP(P) operation which is performed after P does #24 on its first pass cannot be a result of that first pass. Finally, note that a pending wakeup for P is created during P's first pass only if Q does #10 before P does #21, but in this case the pending wakeup will have been cancelled by P at #21 by the time P reaches #24. This completes the proof that Proposition 6 holds for P, the first processor of the chain.

Next we will establish the induction step for the proof of Proposition 5. Suppose the four statements in Proposition 5 hold for all leaders in the chain up to and including Processor R, and let S be the follower of R. (We refer here to a specific appearance of R in the chain, and to its follower on that particular appearance.) Assume S is also a leader (if not there is nothing more to prove) and let T be the follower of S. We want to prove now that Proposition 5 holds for Processor S.

Before beginning with this part of the proof, we will establish the following useful fact. Let U be a processor making an appearance in the current chain, and suppose that U came before S on its previous appearances (if any) in the current chain. Then U's wakeup word will contain zero at the time U does #5 during its present pass. To prove this, note that if U is making its first pass in the current chain, the conclusion follows directly from part (5) of Proposition 4. On the other hand, suppose U has made one or more previous passes in the current chain. When U does #3 on its present pass it clears its wakeup word, which will still contain zero when U reaches #5 unless some other processor changes it. By Proposition 4 this could only be done at #8 by a follower of U from a previous pass in the current chain. But by the induction hypothesis Proposition 5 holds for Processor U during its previous passes, and part (3) of Proposition 5 shows that a follower of U from a previous pass would have already done #8 by the time U completed that pass. Hence U's wakeup word is not altered between the times U does #3 and #5 on its present pass, and consequently contains zero when U does #5.

Now let us proceed with the proof that Proposition 5 holds for Processor S. As the first step in this proof, we will show that S eventually does #15. Since Proposition 5 holds for Processor R, we know that R eventually does #19 and that S takes the "no" path from #6 and hence eventually does #8. We now consider two cases similar to those

introduced for Processors P and Q on page 7-27. In Case I we assume that S does #8 before R does #19. Note that R's wakeup word contained zero when R did #5, by the argument in the previous paragraph. Furthermore that wakeup word will remain unchanged until S does #8, since any previous follower of R must have already done #8 before R could begin its present pass (by part (3) of Proposition 5). Hence S finds zero in R's wakeup word at #8, takes the "yes" path from #9, and performs the SLEEP operation at #11. At #8 S also placed its own number in R's wakeup word, which will remain unchanged thereafter until R does #19. Thus R finds S's number in its wakeup word at #19, takes the "no" path from #20, and performs the WAKEUP(S) operation at #23. After R does #23 S can get past #11, so S is sure to reach #15 eventually in Case I.

For Case II we assume that R does #19 before S does #8. As noted above, R's wakeup word contained zero when R did #5, and will remain unchanged thereafter until (in this case) R does #19. Hence R finds zero in its wakeup word at #19, takes the "yes" path from #20, and performs the SLEEP operation at #21. At #19 R also placed its own number in its wakeup word, so when S does #8 it finds R's number in R's wakeup word, takes the "no" path from #9, and performs the WAKEUP(R) operation at #10. Note that no other WAKEUP(R) could have occurred in the meantime, because then the wakeup by S would still be pending when R finally reached #24, contradicting part (4) of Proposition 5. Hence

R cannot reach #22 until after S has done #10, which of course occurs after S places its number in R's wakeup word at #8. Thus R finds S's number in its wakeup word at #22. We rule out the possibility of another processor changing R's wakeup word by the same reasoning used several times already: Any follower of R left over from a previous pass by R must have done #8, and hence be past the point of changing R's wakeup word, before R started its present pass (by part (3) of Proposition 5), and Proposition 4 admits no other way for R's wakeup word to be altered. We conclude that R will perform the WAKEUP(S) operation at #23 in Case II as well as Case I, again allowing S to get past #11. This completes the proof that S eventually does #15.

Our next objective is to prove that S takes the "no" path from #16 and hence eventually does #19. To do this, we must first determine whether a processor which comes after S in the chain can do #15 before S. Let "time Y" denote the first time at which #15 is executed by a processor which has done #5 more recently than R did #5 (i.e., a processor which comes after R in the chain). By Lemma 3 (FLAG)R remains nonzero from the start of the present chain at least until time Y, so any processor which does #5 before time Y takes the "no" path from #6 (except, of course, for P, the first processor in the chain). Let W be any processor which comes after S in the chain and which reaches #8 before time Y, and let V be the leader of W. If V made a pass in the current chain prior to its pass as W's leader, and if it appeared

after R in the chain on that prior pass, time Y would have occurred before the appearance of W in the chain, contradicting our assumption that W reaches #8 before time Y. Hence V must have appeared ahead of S in the chain on any pass prior to its present pass as W's leader. By the argument beginning at the top of page 7-31, then, V's wakeup word contained zero when V did #5 on its present pass. Clearly, V cannot do #19 before W does #8, because W does #8 before time Y. Thus V's wakeup word still contains zero just before W does #8. As a result, W finds zero in V's wakeup word at #8, takes the "yes" path from #9, and performs the SLEEP operation at #11.

The following arguments show that W will not proceed beyond #11 prior to time Y: By part (5) of Definition 5 no pending wakeup for W is left over from a previous chain; by part (4) of Proposition 5 no pending wakeup is left over from a previous pass in which W appeared ahead of R in the present chain, nor will W receive a wakeup while asleep at #11 resulting from such a previous pass; W's follower on its present pass will not do the WAKEUP(W) operation at #10 prior to time Y, because that follower will take the "yes" path from #9 if it gets to #8 before time Y (as proved in the previous paragraph); W's leader V will not do the WAKEUP(W) operation at #23 prior to time Y, because V cannot proceed past #15 before time Y (by definition of time Y). These exhaust the possibilities for W having a wakeup pending or being waked up prior to time Y, proving that W

cannot pass #11 before time Y. Therefore we conclude that any processor which comes after S in the chain cannot pass #11, and hence cannot get to #15, prior to time Y. But time Y does not occur until some processor which comes after R does #15, so clearly S must be the processor which does #15 at time Y. (We have already proved that S eventually does #15.) Now, we know that R has already done #19 by the time S does #15, and that S's follower T must do #5 before S does #15, so by Lemma 3 we see that S does not set (FLAG)R to zero when doing #15. Therefore S takes the "no" path from #16 and will eventually do #19.

In the preceding paragraphs we established that every processor which does #5 before S does #15 (i.e., prior to time Y) must take the "no" path from #6, except for the first processor of the chain. In particular Processor T, the follower of S, must take the "no" path from #6. This is a fact we needed to establish as part of the proof that Proposition 5 applies to Processor S.

As the next part of the proof, we wish to show that S does #19 before T does #12. By Proposition 2 and part (4) of Proposition 5 there is no pending wakeup in effect for T due to a previous appearance in a prior or the present chain. Thus it will suffice to prove that when T goes to sleep at #11, it cannot be waked up except by S at #23. It is clear from the flowchart that before a processor can do the WAKEUP(T) operation, it must find T's number in some

wakeup word. This number must have been put there by T itself, because one processor never puts another's number into any wakeup word and because the wakeup word in question is known to have contained zero when its owner first did #5 in the present chain. Now a processor which does the WAKEUP(T) operation during T's current pass cannot have found T's number in a wakeup word which T referenced only during a previous pass, because this would contradict part (4) of Proposition 5. Hence it must have found T's number where T put it on this pass. But assuming T has not yet done #19, the only wakeup word it could have referenced during its current pass is that of its leader, S, while doing #8. Thus after T goes to sleep at #11, another processor can wake T up only after finding T's number in S's wakeup word. Now by Proposition 4 the only processor which can do this is S itself, since any follower of S besides T must have already done #8 before S's current pass started (in accordance with part (3) of Proposition 5). Therefore, a wakeup by S at #23 is the only possibility for the WAKEUP(T) operation to be performed when T is asleep at #11 (or has not yet reached #11). Note that this proof that a certain processor sleeping at #11 can only be waked up by its leader at #23 does not apply to an arbitrary processor of the chain. The proof depends on the knowledge that both S and T would have been subject to Proposition 5 on any previous appearances in the chain.

As the next step in the proof that Proposition 5 holds for Processor S, we must show that S eventually does #24, but not before T does #8. We have already established that S eventually does #19, and also that T eventually does #8 (since T takes the "no" path from #6). We again consider two cases. In Case I we assume that T does #8 before S does #19. Then S obviously does not do #24 before T does #8, so we only have to show that S really does do #24. Now between the times T does #8 and S does #19, no other follower of S can change S's wakeup word, because by part (3) of Proposition 5 any such follower would already have done #8 before S began its current pass. Thus by Proposition 4 S's wakeup word remains unchanged during the indicated interval, so S finds a nonzero value (namely, T's number) in its wakeup word when doing #19. As a result S takes the "no" path from #20 and eventually does #24, there being no possibility of blocking in this path.

For Case II, we assume that S does #19 before T does #8. Note that S's wakeup word contained zero when S did #5, by the argument beginning at the top of page 7-31. This wakeup word will not thereafter be altered by a prior follower of S, by the same argument as given in Case I. Thus when S does #19 it sets its AC to zero, subsequently taking the "yes" path from #20 and performing the SLEEP operation at #21. Now by part (5) of Definition 5 (and also part (4) of Proposition 5, if S has appeared previously in the present chain), no pending wakeup was in effect for S

when S did #5 on its current pass. Hence S cannot proceed beyond #21 unless some processor now does the WAKEUP(S) operation, or has already done so during S's current pass. Any such processor must have found S's number in some wakeup word which S referenced during its current pass, by the same reasoning given for Processor T on page 7-36. There are two such wakeup words, namely that of R, which S referenced at #8, and that of S itself, which S referenced at #19. Now by Proposition 4 only R could find S's number in R's wakeup word, since any prior follower of R must be past #8 before the start of S's current pass, and no subsequent follower of R can do #8 until R has repeated #3, erasing S's number from its wakeup word. Similarly only T could find S's number in S's wakeup word during the time in question. Hence a wakeup by R at #23 and a wakeup by T at #10 are the only possibilities for a WAKEUP(S) operation to occur during S's current pass. But the wakeup by R is what allowed S to get past #11, so that wakeup is no longer in effect by the time S reaches #21. We conclude that in Case II only a wakeup by T at #10 will permit S to proceed beyond #21.

The above conclusion makes it clear that S will not do #24 before T does #8 in Case II. To show that S will in fact do #24, however, we must still prove that the wakeup by T at #10 actually occurs in this case. Recall that S sets its wakeup word to its own number when doing #19, and T has not yet done #8 at that time (since we are in Case II). Since any prior follower of S must already be past #8, S's

wakeup word can be altered between the time S does #19 and the time T does #8 only if S does #3 (on a new pass) before T does #8. But we showed above that S cannot proceed past #21 until T does the WAKEUP(S) operation at #10 (if it ever does). Thus S's wakeup word does not change during the time in question, so when T does #8 it does indeed set its AC to S's number, take the "no" path from #9, and perform the WAKEUP(S) operation at #10. This completes the proof that S eventually does #24, but not before T does #8.

To complete the proof that Proposition 5 holds for Processor S, we need to show that there is no pending wakeup in effect for S at the time S does #24, and that no processor performs the WAKEUP(S) operation after that time as a result of the pass S has just completed. From the arguments in the last three paragraphs it is clear that the WAKEUP(S) operation must be performed by R before S can pass #11, and will be performed by T only if S does #21. Hence both wakeups must have already occurred (if they ever will) by the time S reaches #24. After S does #24, its wakeup word can be referenced again only by S itself at #19 (on a new pass) or by a new follower of S at #8. In either case S will have already repeated #3, clearing the number it placed in its own wakeup word at #19 during the pass already completed. S also placed its number in R's wakeup word (while doing #8), and if this occurred after R did #19 then the number will still be there after R finishes its current pass. But in this case R's wakeup word will not be

referenced again until R repeats #3, clearing S's number. We conclude that any WAKEUP(S) operation which is performed after S completes its current pass by doing #24 cannot be a result of the pass just completed. Finally, note that a pending wakeup for S is created during S's current pass only if R does #23 before S does #11 or if T does #10 before S does #21. Either pending wakeup will have been cancelled by S (at #11 and #21, respectively) by the time S does #24. Thus no wakeup is pending for S when S does #24. We have now proved that Proposition 5 holds for Processor S, given that S has a follower and that the proposition holds for all processors in the present chain up to and including the leader of S. This, by induction, completes the proof of Proposition 5.□

7.10 COMPLETION OF THE CORRECTNESS PROOF.

The proof of Proposition 5 has been difficult and time-consuming, but it is a powerful result which tells a great deal about the interactions between processors of a particular chain, provided the system was in the rest state prior to the beginning of that chain. For example, by part (2) each leader does #19 before its current follower does #12. From this is is clear that no two processors of the chain can be in mastermode at the same time. In addition, since the leader/follower relationship is established by the order in which the processors do #5, it is clear that

processors enter mastermode in exactly that order. By part (3) of Proposition 5, each leader in the chain eventually does #24. That is, a processor which begins a pass through mastermode and which has a follower on that pass must eventually complete its pass and return to normalmode. However there may be a processor (namely, the last processor of the chain) which makes a pass without having a follower, and which is therefore not subject to Proposition 5. The next proposition will show that this processor also must eventually do #24, thus establishing that the algorithm has the required freedom from deadlock. Before stating the next proposition, we note one further consequence of Proposition 5. From part (2) it is clear that during a chain to which the proposition applies, an execution of #17 occurs between any two executions of #15. Furthermore, executions of #15 in earlier or later chains are separated from those in the present chain by at least one execution of #5 (performed by the first processor in the present or the next chain, as the case may be). Therefore we can conclude that Lemma 4, which was stated on page 7-10 without proof, and Lemma 5, which depended on Lemma 4, are valid during a chain prior to whose beginning the system was in the rest state. We will use this result in proving the following proposition.

Proposition 6. Suppose the system is in the rest state at some time before the beginning of a given chain and after the end of any previous chain. Then each time a processor appears in the given chain, it will eventually do #24. Furthermore, the last processor of the chain (if there is one) will reach #24 via the "yes" path from #16.

Proof. By Proposition 5, each processor in the chain which has a follower on a given pass eventually does #24. However suppose that at some time a processor, say Processor Z, joins the chain by doing #5, and for an arbitrarily long period of time thereafter no more executions of #5 occur. Then Z may never have a follower, so we cannot use Proposition 5 to prove that Z eventually does #24. But in establishing the induction step of Proposition 5 we proved starting on page 7-31 that a certain processor, S, would eventually do #15. This part of the proof did not depend on the fact that S was a leader, but only on the fact that Proposition 5 was valid for all processors in the chain up to and including the leader of S. Hence by the same argument we can show that Z eventually does #15. Now if another processor does #5 after Z does #5 and before Z does #15, it becomes Z's follower, allowing us to apply Proposition 5 to Z and thus state that Z eventually does #24. On the other hand, suppose that when Z does #15, there have been no executions of #5 since the time Z did #5. Clearly this is the case if and only if Z is the last processor of the chain. By Lemma 5, Z will set (FLAG)R to

zero when it does #15 in this case. As a result Z will take the "yes" path from #16 and will eventually do #24. This establishes the proposition. □

We have now proved everything we would like to prove about the processors of a chain prior to whose beginning the system was in the rest state. In particular, we have shown with Propositions 5 and 6 that the processors of such a chain are granted mutually exclusive first-come-first-served access to mastermode and that any processor which begins a pass through mastermode eventually completes that pass. Our only remaining task, which we undertake in the next proposition, is to prove that the system is in the rest state just prior to the start of every chain.

Proposition 7. Suppose the system is in the rest state at some time before the beginning of a given chain which terminates (i.e., has a last processor) and after the end of any previous chain. Then at some time between the end of the given chain and the beginning of the next chain the system will again be in the rest state.

Proof. Let "Chain C" denote the terminating chain referred to in the statement of the proposition, and let time T1 be a time prior to the start of Chain C (i.e., between the end of the previous chain and the beginning of Chain C) at which the system was in the rest state. Let Processor Z be the last processor of Chain C, and let T2 be the time immediately after the end of Chain C. (That is,

time T2 occurs after Z does #15 and before any processor executes any other step of the algorithm.) Let Processor R be any processor which appears ahead of Z in Chain C (if there is any such processor). Let Processor W be any processor which does not appear in Chain C (if there is any such processor). We will prove the proposition by showing that the system is in the rest state at time T2. To do this, we will consider one by one the five conditions listed in Definition 5 which must be met when the system is in the rest state, and will prove that each condition is satisfied at time T2 by each of Processors R, W, and Z.

(1) If any processor is between #3 and #5, its wakeup word contains zero.

If Processor R is between #3 and #5 at time T2 (i.e., is getting ready to join a new chain which follows Chain C), we know that it cleared its own wakeup word at #3. By part (3) of Proposition 5, any follower of R in Chain C will have already done #8 and hence cannot alter R's wakeup word after R does #3. Proposition 2 rules out the alteration of R's wakeup word by a follower from a previous chain. Hence R's wakeup word must still contain zero at time T2.

Suppose Processor W is between #3 and #5 at time T2. If W did #3 after time T1, it set its wakeup word to zero in doing so. If it did #3 before T1, its wakeup word contained zero at time T1 by part (1) of Definition 5. In either case, W's wakeup word contained zero at some time between T1

and T2. From Proposition 2 and the fact that W does not appear in Chain C, it is clear that W's wakeup word cannot be set to a nonzero value between time T1 and time T2. Hence W's wakeup word must still contain zero at time T2.

Finally, note that Processor Z cannot be between #3 and #5 at time T2, since by definition Z is between #15 and #16 at T2. We conclude that if any processor is between #3 and #5 at time T2, its wakeup word contains zero at that time.

(2) The right half of the shared flag word contains zero.

It follows from Lemma 5 that Processor Z set (FLAG)R to zero when it did #15. Hence (FLAG)R contains zero at time T2.

(3) There are no processors between #5 and #15 or between #16 and #23.

We know that R's follower cannot pass #11 until R does #23. (We established this fact when proving Proposition 5.) But we know from part (2) of Proposition 5 that R's follower (and indeed every processor of Chain C) must have already done #15 at time T2. Hence R cannot be between #5 and #15 or between #16 and #23 at T2.

Since Processor W does not appear in Chain C, it is clear that W cannot be between #5 and #15 at any time between T1 and T2 inclusive. By part (3) of Definition 5, W

was not between #16 and #23 at T1. If it was between #15 and #16 at T1, it cannot be between #16 and #23 at T2 because of part (4), Definition 5. In short, W is not between #5 and #15 or between #16 and #23 at time T2.

Finally, note that Z is between #15 and #16 at T2. We conclude that at time T2, there are no processors between #5 and #15 or between #16 and #23.

(4) Any processor between #15 and #16 will take the "yes" path from #16.

From part (2) of Proposition 5 it is clear that R is not between #15 and #16 at T2.

Suppose W is between #15 and #16 at T2. Then it must have been there throughout the interval from T1 to T2, since it could not do #15 between T1 and T2 without appearing in Chain C. But if W was between #15 and #16 at T1, it will take the "yes" path from #16, by part (4) of Definition 5.

Processor Z is definitely between #15 and #16 at time T2, and by Proposition 6 we know it will take the "yes" path from #16. We conclude that every processor which is between #15 and #16 at time T2 will take the "yes" path from #16.

(5) No pending wakeups are in effect.

As noted above, R must have already done #23 at time T2. Hence there cannot be a pending wakeup in effect for R

at time T2, because this would contradict part (4) of Proposition 5.

By part (5) of Definition 5 no pending wakeup was in effect for W at time T1, and by Proposition 2 the WAKEUP(W) operation is not performed between T1 and T2. Hence there can be no pending wakeup in effect for W at time T2.

Finally, let us consider whether there can be a wakeup pending for Z at time T2. If Z appears more than once in Chain C, then after each pass prior to its last one there will be no pending wakeup in effect for Z, by part (4) of Proposition 5. If Z appears only once in Chain C, then its last pass is also its first pass, at the start of which there was no wakeup pending for Z by part (5) of Definition 5. Thus in either case there is not a pending wakeup in effect for Z when it begins its final pass in Chain C. Since Z has no follower on its final pass, the WAKEUP(Z) operation can only be performed during that pass by Z's leader at #23, and if a pending wakeup is generated thereby (i.e., if Z has not reached #11 at the time), that pending wakeup will be cancelled by Z at #11. Thus there can be no pending wakeup in effect for Z at time T2. We conclude that at time T2 no pending wakeups are in effect.

We have now shown that all the conditions for being in the rest state are satisfied at time T2, completing the proof of the proposition.□

The application of Proposition 7 depends on the fact that the system is in the rest state sometime before the beginning of the first chain. The proof of this is trivial if we pick a time before any processor has started executing the algorithm and recall that zero in the shared flag word and the absence of pending wakeups are specified initial conditions assumed to hold prior to processor activation. Given that the system is initially in the rest state, a simple inductive argument using Proposition 7 establishes that the system is in the rest state at some time between any two successive chains. As a result, we can state that Propositions 5 and 6 apply to every chain. As noted earlier, these propositions show that the algorithm provides the required mutually exclusive first-come-first-served access to mastermode and freedom from deadlock. To rephrase this more emphatically, we have now rigorously proved that the flowchart algorithm provides a correct solution to the mastermode/normalmode problem, assuming that the shared memory locations are protected from extraneous references not connected with the algorithm and that the SLEEP and WAKEUP operations only occur where shown in the flowchart. The latter assumption is considered further in the next section.

7.11 THE ASSUMPTION OF SECTION 7.2 REVISITED.

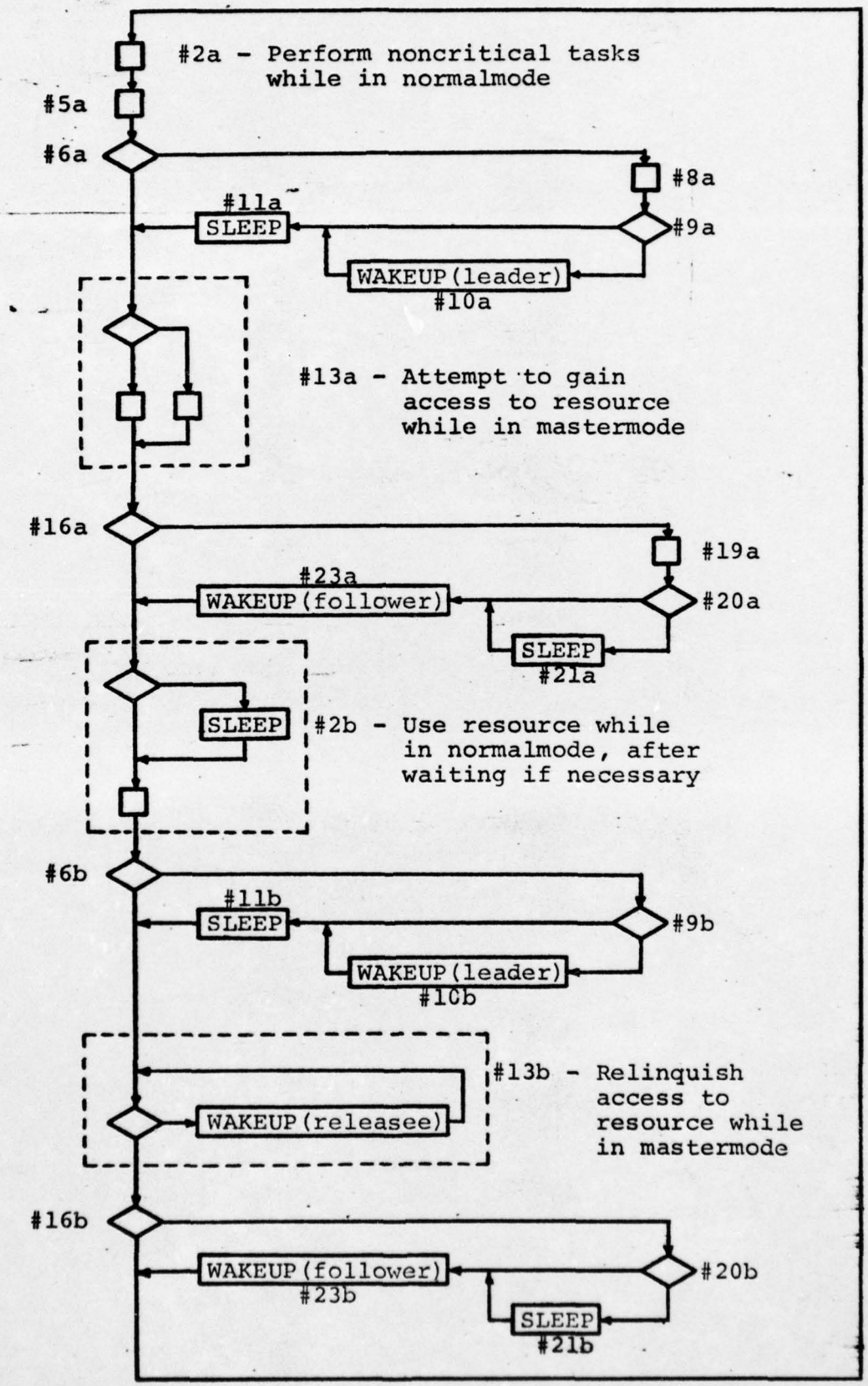
The correctness proof just completed was based on the assumption that SLEEP and WAKEUP operations only occur where explicitly shown in the flowchart. We must now show that additional SLEEP and WAKEUP operations of a specific sort will not interfere with the correct operation of the MASTERMODE/NORMALMODE mechanism. These additional operations are the ones which the processors perform while competing for access to a general shared resource, as described in Chapter 4. The scheme for such competition developed in Chapter 4 used the same SLEEP/WAKEUP mechanism to control processor delay that we have now used to implement the MASTERMODE/NORMALMODE operations. Before contenting ourselves that these operations work correctly, we must consider possible interaction between the two uses of the SLEEP/WAKEUP mechanism.

Recall from Chapter 4 that a processor attempting to use a shared resource might delay itself with the SLEEP operation, in which case it would later (or perhaps earlier) receive a wakeup from a processor relinquishing access to the resource. For convenience we will refer to the relinquishing processor as the "releaser" of the processor it awakens, and to the processor which receives the wakeup as the "releasee". On page 7-51 is shown a simplified flowchart (Figure 7-1) representing the cyclic use of a shared resource by a processor which uses the

MASTERMODE/NORMALMODE operations to enforce the required mutual exclusion. The steps in this flowchart have been given the same numbers as the corresponding steps in the flowchart of the MASTERMODE/NORMALMODE algorithm. Since one cycle of shared resource access involves two passes through mastermode, each step of the MASTERMODE/NORMALMODE algorithm appears twice in Figure 7-1. Thus #6a identifies step #6 of the algorithm in the processor's first pass through mastermode (when it gains access to the resource) and #6b identifies step #6 in the processor's second pass (when it relinquishes access to the resource). Before continuing the reader is advised to work out the connections among the flowchart in Figure 7-1, the general programs on pages 4-7 and 4-9, and the flowchart of the MASTERMODE/NORMALMODE algorithm in Figure 7-2.

Although many steps have been omitted in Figure 7-1 to simplify the flowchart, all SLEEP and WAKEUP operations are shown. Our correctness proof guarantees that for each SLEEP operation performed by a processor as part of a MASTERMODE or NORMALMODE operation, one and only one WAKEUP operation will be (or has been) directed at that processor by some other processor which, as we know, is either its leader or its follower. Thus a SLEEP at #11a is matched by a WAKEUP at #23a performed by the leader of the sleeping processor, and a SLEEP at #21a is matched by a WAKEUP at #10a performed by the follower of the sleeping processor. Similar remarks apply to steps #11b, #23b, #21b, and #10b. Furthermore,

Fig. 7-1. Cyclic use of a shared resource. Page 7-51



assuming that proper strategies have been chosen to accomplish the resource sharing, we know that a processor which delays itself with a SLEEP operation while gaining access to the resource will sooner or later be waked up by its releaser when the latter relinquishes the resource. Thus for each SLEEP operation performed at #2b there is a matching WAKEUP operation performed at #13b by the releaser of the sleeping processor.

The condition just described of having each SLEEP operation matched by exactly one WAKEUP operation is clearly necessary for correct system behavior. Unfortunately it not by itself a sufficient condition, because there is a way for WAKEUP operation to "get lost". Suppose some processor performs a WAKEUP(P) operation at a time when Processor P is not asleep, i.e., before P has performed the SLEEP operation which matches the given WAKEUP operation. Then a pending wakeup goes into effect for P which will keep P from being delayed when it finally does perform the SLEEP operation. But suppose another WAKEUP(P) operation is performed by some processor in the meanwhile. The SLEEP/WAKEUP mechanism does not allow a processor to "remember" more than one pending wakeup, so the second WAKEUP(P) operation will have no effect. Now, however, Processor P has two SLEEP operations to perform, one matching each WAKEUP(P) operation. When P performs the first of these SLEEP operations it will cancel its pending wakeup and continue to run, but when it performs the second it will go to sleep and never wake up, since the

matching WAKEUP(P) operation has already occurred and has been ignored. This is clearly not the behavior desired of Processor P.

From the correctness proof in the preceding sections it is clear that the improper behavior just described would not occur if the SLEEP and WAKEUP operations were limited to those which implement the MASTERMODE/NORMALMODE operations. Furthermore, if the latter operations did not use the SLEEP/WAKEUP mechanism, the improper behavior described above would not arise from the use of SLEEP and WAKEUP operations in resource sharing. To see this, consider that a processor only goes to sleep and receives a matching wakeup as a result of trying to gain access to the shared resource, if the SLEEP/WAKEUP operations associated with MASTERMODE/NORMALMODE are disregarded. Clearly the processor cannot try to gain access to the resource a second time until it has passed the SLEEP operation on the way to its first access, and so there is no possibility of a second wakeup prior to that time.

In short, the SLEEP/WAKEUP operations have been shown to work correctly both for resource sharing and for enforcement of mutual exclusion when the two uses are considered separately. Our real concern is with interaction between the two uses that might lead to the improper behavior described above. To determine how such interaction might occur, let us consider for each SLEEP operation in

Figure 7-1 how early the matching WAKEUP operation might occur. A processor about to sleep at #11a will sooner or later receive a wakeup from its leader, and this clearly cannot happen before the processor about to sleep acquires a leader by doing #5a. Thus a processor may receive a wakeup before it reaches #11a, resulting in the creation of a pending wakeup, but such a pending wakeup cannot be created before the processor even reaches #5a. A processor does not sleep at #21a unless it does #19a before its follower, which will later wake it up, does #8a. Thus a pending wakeup for a sleep at #21a cannot be created until the processor about to sleep at least does #19a. The same considerations apply to the SLEEP operations at #11b and #21b. Finally, a processor which is about to sleep at #2b in order to wait for the shared resource cannot receive its matching wakeup until it has at least done #13a, because the wakeup results from changes in shared state information which the processor made while trying to gain access to the resource at #13a.

From the above considerations it is clear that there is only one possibility for the improper behavior which would result if a processor received two wakeups prior to performing the SLEEP operation matching either of them. Specifically, a processor which has done #19a and is about to sleep at #21a may receive a wakeup from its follower (when the follower does #10a) and may receive a wakeup from its releaser (when the releaser does #13b) prior to sleeping at #21a.

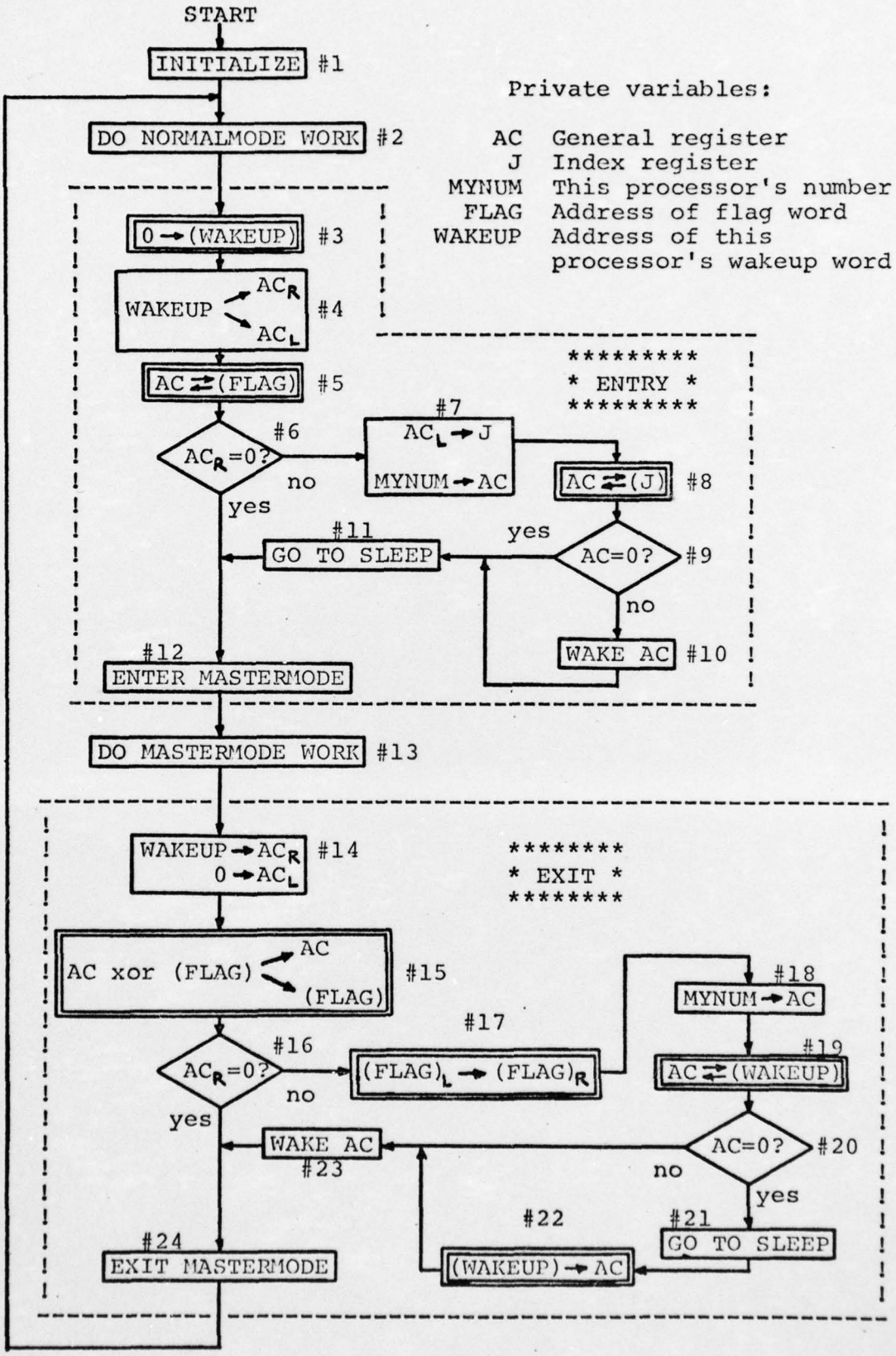
In view of all the work we have done up to this point, it is fortunate that we can rule out the possibility just described. We can do so because the WAKEUP operation performed by a releaser at #13b is done in mastermode, and hence cannot occur while the releasee is in mastermode. We proved starting on page 7-35 that a follower cannot proceed beyond #11, and hence cannot enter mastermode, until its leader has done #23. Thus a releaser cannot enter mastermode and wake up its releasee at #13b while the releasee is between #19a and #21a. To state this more carefully, we have shown that Proposition 5 will not be violated due to a bad SLEEP/WAKEUP interaction unless it has already been violated by the appearance of two processors in mastermode simultaneously. But the latter violation could only result from a bad SLEEP/WAKEUP interaction, since such an interaction is the only possibility we failed to consider when proving the proposition. We conclude that a violation of Proposition 5 can never occur in the first place. Now we are really finished with the proof that we have found a correct solution to the mastermode/normalmode problem. For completeness we will state the outcome of the arguments of this section in a final proposition.

Proposition 8. Although the correctness proof for the MASTERMODE/NORMALMODE algorithm depended on an assumption that SLEEP and WAKEUP operations only occur where shown explicitly in the flowchart of Figure 7-2, the conclusions of the correctness proof remain valid when additional SLEEP

and WAKEUP operation occur in accordance with the general procedure for resource sharing given in Chapter 4.

Proof. Given above.

Fig. 7-2. Flowchart of solution algorithm. Page 7-57



CHAPTER 8

SYNCHRONIZING INDEPENDENT GROUPS OF PROCESSORS

8.1 INTRODUCTION.

In Chapter 4 several processor coordination problems were solved using a basic processor delay mechanism (the SLEEP/WAKEUP mechanism) and a basic mutual exclusion mechanism (the MASTERMODE/NORMALMODE mechanism). At that time it was just assumed that the MASTERMODE and NORMALMODE operations could be implemented efficiently with available lower-level operations. This assumption has now been justified by the algorithm and correctness proof given in Chapters 6 and 7.

Although the coordination problems in Chapter 4 were solved rather easily, the solutions had to include detailed specification and manipulation of the shared variables and processor queues involved. The need for this detail in the solutions arose because of the low level at which the problems were solved (i.e., because of the functional simplicity of the available coordination mechanisms). Such low-level coordination is appropriate in system programming situations, such as the development of computer operating

systems, because the required coordination mechanisms can be made to operate with the high efficiency needed in such situations.

On the other hand, being able to give a simple description of a desired solution is probably more important than achieving maximum efficiency to an applications programmer working in a high-order programming language. This suggests that we may want to develop a higher-level abstraction of our computing system in which we can solve coordination problems without worrying about the small details of shared storage allocation and processor queue manipulation. That is, we may want to devise a powerful "processor synchronization language" and implement it using the coordination mechanisms available at our present level of abstraction. Several languages of this kind have been proposed, for example Presser's extension of Dijkstra's semaphore operations (Pr75) and Brinch Hansen's "critical region" language (Br72).

It is not our intention to develop a new high-level processor synchronization language. We <u>are</u> interested, however, in determining whether the low-level coordination mechanisms described in previous chapters are powerful enough to implement such languages. In Chapter 4 we demonstrated the use of SLEEP, WAKEUP, MASTERMODE, and NORMALMODE operations to solve arbitrary processor synchronization problems involving the allocation of a

single shared resource. Most proposed high-level synchronization languages, however, allow different groups of processors to compete for several independent shared resources without interference. We will show in this chapter that such problems can also be solved using the low-level approach described in Chapter 4.

8.2 ALLOCATION OF INDEPENDENT SHARED RESOURCES.

Suppose that several readers and writers of the type described in Chapter 4 are competing for access to a data file, and suppose that at the same time several other readers and writers are competing for access to a different and independent data file. We will refer to the two groups of processors as Group A and Group B. We could provide for the proper behavior of all the processors with programs similar to those developed in Chapter 4. Of course, the shared variables used for communication among the processors would have to be different for the two groups. That is, the shared arrays used to implement the waiting lines and the shared variables WRITING and READERS would have to be duplicated, so that each group would have a separate set of those shared locations.

There is one serious disadvantage to a solution of the form just described: A reader or writer which enters mastermode to gain or relinquish access to its data file will exclude from mastermode the other processors not only

AD-A032 803 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OHIO SCH--ETC F/G 9/2
EFFICIENT MULTIPLE PROCESSOF COORDINATION.(U)
NOV 76 B E BAKER

UNCLASSIFIED DS/EE/76-1 NL

3 OF 3
AD A032803

END

DATE FILMED
2-77

of its own group but also of the other group. Thus for example a reader in Group A might be delayed from gaining access to the file belonging to Group A while a writer in Group B relinquishes access to that group's file. Since the two groups of processors use disjoint sets of shared locations to control access to their respective data files, such mutual exclusion between groups is unnecessary, and hence the delay which it might cause is undesirable and ought to be avoided.

To prevent the unnecessary delay just described, we want two processors to be able to enter mastermode at the same time provided they belong to different groups. Fortunately the MASTERMODE/NORMALMODE algorithm presented in Chapter 6 makes this possible with no change in the basic form of the algorithm. Recall that the algorithm associates with each processor a shared location referred to as the processor's wakeup word, and provides an additional shared location (the shared flag word) which is used symmetrically by the various processors. The desired noninterfering use of the MASTERMODE/NORMALMODE mechanism by independent groups of processors can be achieved by simply providing each group with a different shared flag word.

In terms of the algorithm's data base this is accomplished by initially storing the address of a particular shared location in the private memory location called FLAG of each processor in Group A, and by storing the

address of a different shared location in each private FLAG location of the Group B processors. (If this is not clear, review the data base description starting on page 6-8.) The shared location whose address is stored in the FLAG locations of the Group A processors becomes the shared flag word for that group, and similarly for Group B. A review of the MASTERMODE/NORMALMODE algorithm makes it clear that with different groups of processors using different shared flag words, a processor in one group never references a shared flag or wakeup word of another group. Thus processors in one group enter mastermode and return to normalmode independently of processors in a different group.

In Chapter 4 we described an ALGOL implementation of the solution to one of the reader/writer problems. Although not used or explained in Chapter 4, a simple method was provided to allow coordination of independent groups of readers and writers. This method consists of a capability to specify for each processor an offset for its references to the shared memory segment. Recall that each reader and writer invokes the INITIALIZE procedure near the start of its program with a call of the form INITIALIZE(MYNUM,k), where MYNUM is the number of the calling processor. The second argument k is a nonnegative integer which will be added to the processor's private storage locations FLAG and WAKEUP. Recall that these locations are part of the data base for the MASTERMODE/NORMALMODE algorithm and contain the addresses of the shared flag word and the processor's wakeup

word, respectively. The addresses contained in these locations are established by the INITIALIZE procedure and are not changed thereafter. The processors can be divided into independent groups by specifying a different offset k for the processors of each group. The offset for each group must be chosen so that the shared locations used by one group do not overlap with those used by another group.

As we have just noted, the value of k specified by a given processor is used as an offset for the implicit memory references which that processor makes to shared flag and wakeup words in performing the MASTERMODE and NORMALMODE operations. Therefore processors using different offsets (i.e., processors in different groups) will not interfere with one another's MASTERMODE and NORMALMODE operations. The offset k can also be used by each processor when establishing pointers to the shared memory locations which the processor will reference explicitly using the SET, ASSIGN, TRULY, and VALUE.OF procedures. As a result, processors in different groups will use different locations for the shared variables which appear in their programs (for example the shared arrays representing waiting lines and the shared variables WRITING and READERS in the reader/writer programs). This means that corresponding processors in different groups can use programs which are identical except for the offset used when calling the INITIALIZE procedure and when establishing pointers to shared variables.

Specifying different offsets for different groups of processors is not a very sophisticated approach inasmuch as it requires a static allocation of shared memory locations for each group. It is not hard to envision more elaborate schemes for partitioning the shared locations used by various groups of processors. Nevertheless the simple approach described above is adequate to illustrate the coordination of multiple groups of processors and to demonstrate the potential capabilities of the MASTERMODE/NORMALMODE algorithm. We will explore these capabilities further in the next section by considering a processor coordination problem belonging to a different family from the problems discussed in Chapter 4.

8.3 PRODUCERS AND CONSUMERS WITH A POOL OF BUFFERS.

8.3.1 Statement of the Problem.

In order to illustrate the coordination technique described in the preceding section, we will now state and solve a problem involving independent groups of processors. The problem we will consider belongs to a family of processor synchronization problems known as the producer/consumer problems. On page 2-2 we introduced a simple problem of this type involving two processors called the producer and the consumer. Recall that the producer created packets of data and stored them in a buffer, and the consumer retrieved the data packets from the buffer and

subjected them to further processing. We will consider an extension of this problem in which packets are produced by an arbitrary number of producers for consumption by an arbitrary number of consumers. In addition we will assume that instead of just one buffer there is a pool of identical buffers available for passing packets between the producers and the consumers. Thus it will be possible for various processors to be producing, depositing, retrieving, and consuming data packets simultaneously.

Several versions of the above problem are possible depending on various processor and packet ordering requirements. In the version we will consider, we do not require that packets be consumed in the same order in which they were produced, and we do not care which consumer retrieves a packet produced by a particular producer or stored in a particular buffer. We merely want the producers to keep producing packets and filling buffers and the consumers to keep emptying buffers and consuming packets as fast as possible. Thus when a producer has a packet ready, it may be allowed to fill any available buffer. It will have to be delayed only if all buffers are currently being filled or emptied or are already full. Similarly a consumer which is ready to retrieve a packet will have to be delayed only if no full buffers are available. If several producers are waiting for an empty buffer, or if several consumers are waiting for a full buffer, we require that the next empty or full buffer be assigned to the producer or consumer which

has been waiting longest, to avoid the possibility of some processor having to wait forever.

The producer/consumer problem described up to now involves a single group of producers and consumers all of which interact with one another. Now let us suppose that there are several independent groups of producers and consumers working simultaneously, each group having its own separate pool of buffers. Solving the problem in this form will allow us to try the approach described previously for coordinating independent groups of processors. Later in this chapter we will further complicate the producer/consumer problem by introducing a new kind of processor which can interact with the processors of more than one group. First however we will develop a solution to the producer/consumer problem as it has been presented up to this point.

8.3.2 Processor Strategies.

To solve the producer/consumer problem described above we must first determine the strategies to be followed by the producers and consumers. We will keep different groups of processors separate by offsetting their references to the shared memory segment, and hence the processors' strategies will not have to take into account the existence of other groups. That is, the strategies will be the same as if there were only one group of producers and consumers. Since

the processors may have to wait for an available buffer, we will provide two waiting lines, one for the producers and one for the consumers. Of course we will end up with two such lines for each separate group of processors, but this will come about automatically because of the way the processors' references to shared memory occur. The strategies are now rather easy to determine.

Producer Strategy

(1) Check the buffers in any order to find an empty one. If an empty buffer is found, begin filling it. If there are no empty buffers, join the producer line and wait.

(2) After filling a buffer, if the consumer line is not empty allow the consumer at the head of the line to leave the line and begin emptying the buffer just filled.

Consumer Strategy

(1) Check the buffers in any order to find a full one. If a full buffer is found, begin emptying it. If there are no full buffers, join the consumer line and wait.

(2) After emptying a buffer, if the producer line is not empty allow the producer at the head of the line to leave the line and begin filling the buffer just emptied.

8.3.3 Assignment of Shared Variables.

Now we must determine the system state information required by the producers and consumers to carry out the above strategies. We will distinguish the processors and buffers in an given group by numbering the producers from 1 to NPROD, the consumers from 1 to NCON, and the buffers from 1 to NBUF, where NPROD, NCON, and NBUF are the number of producers, consumers, and buffers, respectively, in the given group. We will implement the waiting lines just as we did in the reader/writer problems in Chapter 4. Thus we provide shared integer arrays P.LINE[-1:NPROD] and C.LINE[-1:NCON] to represent the producer and consumer waiting lines. The first two elements in each array must initially contain zero, indicating that the lines are empty.

It is clear from the strategies in the preceding section that the processors must be able to tell if a given buffer is full or empty. To make this determination possible we will provide two shared Boolean arrays, FULL[1:NBUF] and BUSY[1:NBUF], which will be used as follows. FULL[J] will be true if and only if Buffer J is full, meaning that a producer has finished filling it and a consumer has not yet started to empty it. BUSY[J] will be true if and only if Buffer J is "busy", meaning that it is in the process of being filled or emptied. Thus a consumer tries to find a full buffer by searching for a value of J such that FULL[J] is true, and a producer tries to find an

empty buffer by searching for a value of J such that both FULL[J] and BUSY[J] are false. All elements of the arrays FULL and BUSY must initially be false, indicating that the buffers are empty.

Finally, we note that a processor which has been waiting for a buffer to become available must be able to determine which buffer it is supposed to use, when it is finally allowed to proceed. To make this determination possible we will provide two shared integer arrays, P.USE[1:NPROD] and C.USE[1:NCON]. P.USE[J] will contain the number of the buffer to be filled next by Producer J, and C.USE[J] will contain the number of the buffer to be emptied next by Consumer J. Thus when Consumer J is just on the verge of emptying a buffer (i.e., has proceeded past the conditional SLEEP operation with which it delayed itself if necessary), it must examine C.USE[J] to find out which buffer to use. The value in C.USE[J] was placed there either by Consumer J itself, if it found a full buffer during its search, or by a producer which found Consumer J at the head of the consumer line after filling a buffer. In the latter case Consumer J would have delayed itself by performing a SLEEP operation. Clearly the producer which finds Consumer J at the head of the consumer line must place the number of the buffer it has just filled into C.USE[J] before performing the WAKEUP operation which will allow Consumer J to proceed. This is the first time we have had to be concerned with the exact sequence of operations

performed by a processor while in mastermode, because in the problems solved previously it was never necessary for a processor to examine a shared location while in normalmode.

The six arrays defined above, namely the arrays P.LINE, C.LINE, FULL, BUSY, P.USE, and C.USE, constitute the shared state information needed by the processors to solve the given producer/consumer problem.

8.3.4 ALGOL Programs for the Producers and Consumers.

ALGOL programs which implement the producer and consumer strategies are listed in Appendix C on pages C-2 through C-5. The programs allow for up to four groups of producers and consumers, referred to as Groups A, B, C, and D. Note that each processor will be supplied with input data specifying the group it belongs to and the number of producers, consumers, and buffers in that group. Each producer also reads in a value for the integer variable P.LOOP, which specifies how many times the producer will go through its cycle of producing and depositing a data packet. Thus the number of packets produced may be different for different producers. Each consumer has an integer variable C.LOOP which serves the same purpose.

The external procedures declared in the producer and consumer programs are the same ones which were used in the reader/writer programs and which were described starting on

page 4-23. The REPORT procedure has been modified slightly to permit each processor to include the name of its group when making a report.

The coding of the cyclic portion of the producer and consumer programs is derived directly from the producer and consumer strategies and from the generalized resource allocation programs given in Chapter 4 (pages 4-7 and 4-9). The reader may wish to verify that the programs as coded correctly implement the specified producer and consumer strategies.

8.4 A NEW KIND OF PROCESSOR INTERACTION.

In the synchronization problem solved in the preceding section, each group of processors had its own pool of buffers and there was no interaction at all between processors of different groups. It is not unusual, however, for one processor to require access at various times to a number of different shared resources. One way of describing this situation is to say that the processor in question belongs to different groups of processors at different times. The group it belongs to at a given time consists of all the other processors which are competing for access to the particular shared resource it is trying to use at that time.

We will now consider an extension of the producer/consumer problem which will illustrate the possibility of one processor interacting with the processors of several different groups. The extension involves a new class of processors known as "distributors". A distributor is like a producer in that it periodically creates a packet of data which it stores in a buffer for later retrieval by a consumer. The unique characteristic of a distributor is that it is free to deposit its data packet in a buffer belonging to any of the producer/consumer groups. Thus a distributor may be thought of as a member of different groups of producers and consumers at different times. At a particular time it is a member of the group in whose buffer pool it is waiting to deposit or actually depositing a data packet.

When a distributor is interacting with a given group of processors, it behaves as though it were just another producer in that group. Thus the strategy followed by the distributor is identical to the producer strategy described earlier. The presence of distributors does not change the shared state information required for a solution, but the shared arrays P.LINE and P.USE must be large enough to provide not only for the actual producers of a given group but also for all the distributors which might occasionally join that group. The producer and consumer programs need not change at all for a solution in which distributors are present. However the value of NPROD supplied as input data

to each producer and consumer must account for the existence of distributors. Thus if a certain group contains three producers and there are two distributors which may join the group, the producers and consumers will be informed (by way of the value they read for NPROD) that the group contains five producers. Their operation will not be affected by the fact that two of the "producers" are actually distributors.

Now let us consider how distributors join particular groups of processors. Recall that we divide processors into different groups by arranging for them to use disjoint sets of shared memory locations. Thus when we say that a distributor joins a given group, we mean that it starts using the shared locations associated with that group. Of course it should not do this while it is still interactiing with the processors of another group. That is, a distributor should only change groups when it is at a point in its program corresponding to the "perform private tasks" step of the general program on page 4-4.

Each processor establishes pointers to the shared locations that it is going to reference explicitly. Thus when a distributor wishes to join a new group it can arrange to use such locations associated with the new group simply by redefining its pointers. On the other hand, the shared locations which a distributor references implicitly in performing the MASTERMODE and NORMALMODE operations present a problem, because the distributor cannot directly access

these locations. To solve this problem we will provide a new procedure called NEWGROUP. A call to this procedure has the form NEWGROUP(MYNUM,k), where MYNUM is the number of the calling processor and k is an offset whose function is the same as that of the offset specified when calling the INITIALIZE procedure. Thus a call to NEWGROUP causes the addresses stored in private locations FLAG and WAKEUP of the calling processor to be adjusted by the offset k, so that when the processor subsequently performs MASTERMODE and NORMALMODE operations it interacts with those processors which specified the same offset in calls to INITIALIZE or NEWGROUP.

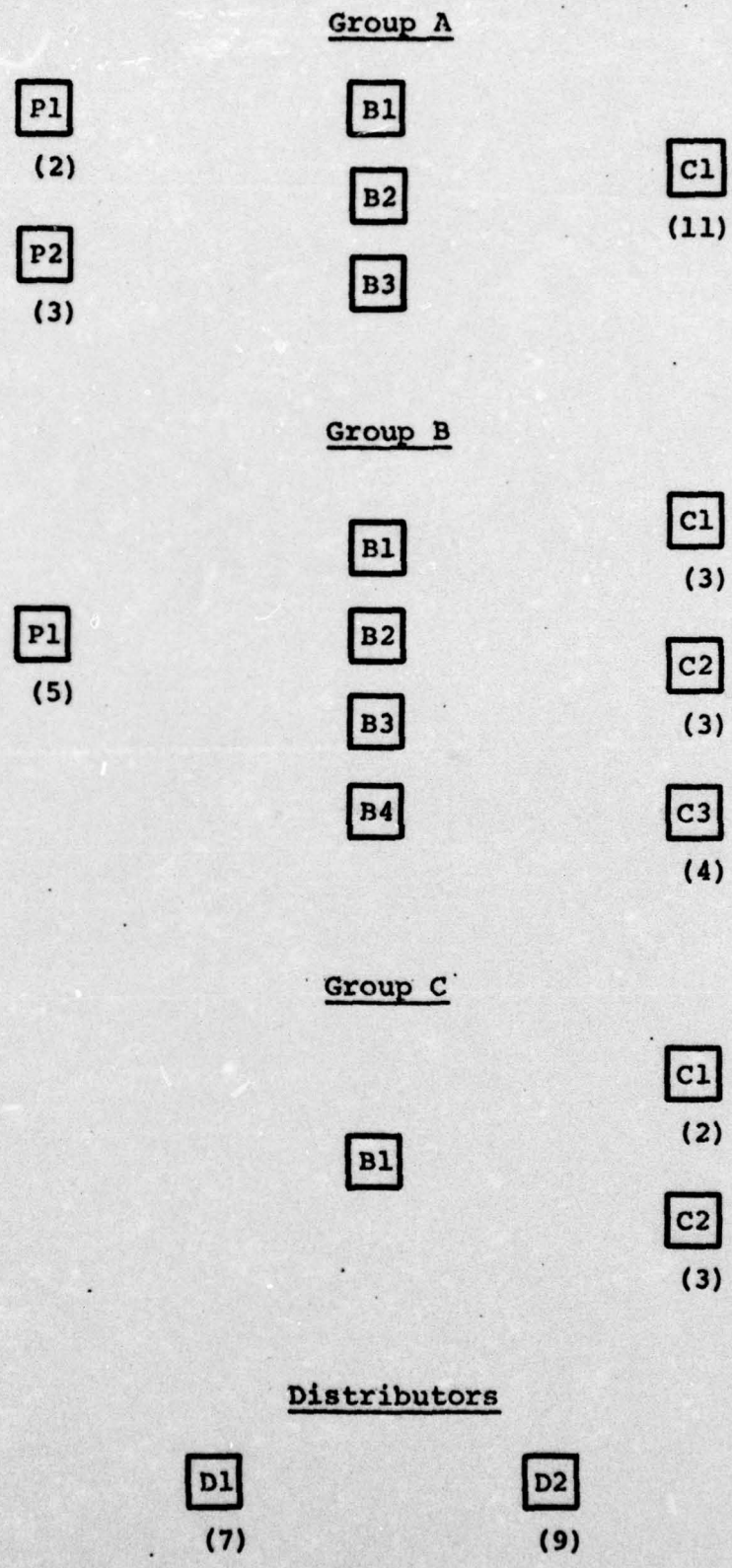
8.5 SOLUTION TO THE PRODUCER/CONSUMER/DISTRIBUTOR PROBLEM.

An ALGOL program implementing a distributor is listed on pages C-6 and C-7 of Appendix C. This program allows a distributor to become a member of Group A, B, or C. Note that the input data supplied to the distributor includes the number of producers, consumers, and buffers in each of those groups. Input values are also supplied for NDIS, the number of distributors, and D.LOOP, the number of packets to be produced by this distributor. The external procedures called by the distributor are the same ones used by the producers and consumers with the addition of

Instead of establishing pointers to shared variables once at the beginning of its program, the distributor defines an internal procedure named SET.POINTERS which it can invoke when changing groups. The cyclic portion of the program is almost the same as that of the producer program. After each produce/deposit cycle the distributor joins a new group of producers and consumers by calling the SET.POINTERS procedure to redefine its pointers to shared variables and by calling the NEWGROUP procedure so that it will interact with processors in the new group when subsequently performing MASTERMODE and NORMALMODE operations. For the sake of simplicity the distributor has been programmed to deposit its data packets cyclically in the buffers of Groups A, B, and C in that order.

Following the program listings in Appendix C is a listing of output data for an experimental run with three groups of producers and consumers and two distributors. The configuration of the processors for this run is shown in Figure 8-1 on the next page. In each group of processors the producers are shown on the left, the buffers in the middle, and the consumers on the right. The distributors at the bottom belong to each group in turn. Note that Group C has no producers, so that only the distributors deposit data packets in the buffer of that group. The number in parentheses under each processor indicates the number of packets produced or consumed by that processor. To ensure that all the processors terminate their execution normally,

Fig. 8-1. Experimental processor configuration.



these numbers have been chosen so that the number of packets produced in each group is equal to the number consumed in that group. The ambitious reader may wish to verify that the results listed in Appendix C reflect the desired behavior of the processors.

8.6 A FINAL CLARIFICATION.

Before concluding our discussion of the programs developed to illustrate multiple processor coordination, we wish to eliminate a possible source of confusion. The alert reader has probably noticed that we did not discuss the effect of independent groups of processors on the SLEEP and WAKEUP operations. The very alert reader has probably noticed that when we establish pointers to shared variables in all of the sample programs, we skip the first 2N+1 locations in a given group's block of shared locations, where N is the total number of processors in the group. N+1 of these unexplained locations are used for the shared flag and wakeup words needed to implement the MASTERMODE and NORMALMODE operations. We did not describe this scheme earlier because it is only one of many possible ways of allocating those shared locations and it has no direct bearing on the nature of the solutions.

The remaining N unexplained locations contain a mapping between the processor numbers of a given group and the "logged-in job numbers" which must be used in the monitor

call that actually performs the wakeup function. This mapping is established for a given processor when it calls the INITIALIZE or NEWGROUP procedures and is used thereafter when any processor sends a wakeup to the given processor. This explains why the wakeup mechanism is not deceived by the occurrence of WAKEUP(k) operations in which the same value of k refers to processors in different groups. Of course the SLEEP operation does not have a similar problem, because it has no argument and only affects the processor which performs it.

CHAPTER 9

CONCLUSION

9.1 SUMMARY.

The producer/consumer/distributor problem solved in the last chapter is the final example we will give of the use of efficient low-level operations to coordinate the activity of interacting processors. The programs in these sample problems, which were developed by following the general procedure described in Chapter 4, were not intended to be "practical" examples of efficient processor synchronization solutions. Indeed, if efficiency were our only concern we would have written the programs in assembly language rather than in ALGOL. Using programmed monitor calls to control processor delay and coding the solutions in a high-order language should be considered merely as convenient techniques for simulating the multiprocessor environment in which low-level coordination is needed and for illustrating such coordination in a straightforward way. Hopefully the reader has not lost sight of the objective we set for ourselves in Chapter 1, which was to determine how one can efficiently organize a complex computing system containing

one or more physical processing units whose design does not include explicit processor synchronization capabilities. The result of our investigation may be summarized in the following advice which we would offer to the computer system architect or programmer faced with the task of organizing such a computing system.

(1) Resolve to abide by the principles of a level-structured system design philosophy. This requires that the hardware and software of the system be organized in a series of "levels of abstraction" in which each abstraction is developed from the level below it and represents a virtual computing system which has some desirable property or capability not available at lower levels. Attempt to reach a level of abstraction as early as possible in which it is no longer necessary to account for the exact configuration of the underlying physical system. The objective of such a design philosophy is to develop a system which is "robust" in the sense defined by Dijkstra (Di71). That is, creating an abstraction which is independent of the system's physical configuration will give us some immunity against the effects of accidental or intentional changes in that configuration. Furthermore the carefully organized stepwise development required in a level-structured design will make it easier for us to establish the correctness of that design.

(2) For the first abstraction of the physical system, implement a collection of virtual processors which can execute the machine instructions of the physical processor(s) and which can also perform efficient operations to suspend their execution when necessary. That is, provide the virtual processors with a delay mechanism such as the SLEEP/WAKEUP mechanism described in Chapter 2. The implementation of this initial abstraction will necessarily depend on the specific nature of the physical system from which the abstraction is developed.

(3) At the next level of abstraction, provide processors which are able to perform all lower-level operations and which are also able to enforce mutual exclusion within "critical sections" of their programs. The implementation of the mutual exclusion mechanism will depend upon the processor delay mechanism developed previously and the other operations available to the processors, but will not be influenced by the specific configuration of the underlying physical system. Expend as much effort on the creation of this abstraction as is necessary to devise a mutual exclusion mechanism with all the desirable characterisics outlined in Chapter 5 and to become convinced of the correctness of the mechanism's operation. The main achievement of our investigation has been to demonstrate, through the development of the MASTERMODE/NORMALMODE mechanism, the feasibility of creating such an abstraction based upon a simple processor delay mechanism and the

machine instruction set of a particular physical processing unit.

(4) Develop a higher level or levels of abstraction in which it is possible to perform operations which synchronize processor activity without worrying about small details such as shared storage allocation and processor queue manipulation. The implementation of such "high-level coordination" operations will depend upon the lower-level processor delay and mutual exclusion mechanisms developed previously. Thus the system designer should have in mind a systematic approach to low-level coordination such as the one illustrated in Chapters 4 and 8. Our investigation did not extend to the level of abstraction just described. We did, however, solve processor synchronization problems of sufficient complexity to suggest that our proposed low-level mechanisms are suitable for implementing desirable high-level coordination operations.

9.2 SUGGESTIONS FOR FURTHER WORK.

In the course of this investigation a number of questions have occurred to the author which suggest areas for further study under the general heading of "low-level processor coordination". The most interesting of these questions are listed here.

(1) The MASTERMODE/NORMALMODE algorithm presented in Chapter 6 was based on the machine instruction set of the DECsystem-10 KI10 CPU. Can mutual exclusion mechanisms with the same desirable properties be developed using the machine instructions available in other modern computer systems? If so, how will the algorithms differ from the one given in Chapter 6?

(2) Suppose that instead of being given a physical processor to work with, we are designing a new one and can specify the machine instructions we want it to have. What instructions should we include to facilitate the solution of processor synchronization problems? What can we say in general about the characteristics of instruction sets with which it is possible to solve the mastermode/normalmode problem stated in Chapter 5? Are the read-modify-write, register/memory exchange, and halfword data transfer instructions which we used to implement a solution the most suitable types of instructions for processor coordination?

(3) The proof of correctness given in Chapter 7, while believed to be completely rigorous, was not based on any formal theory of program correctness. In addition it was quite long and difficult. Can formal or automated theorem-proving techniques also be used to establish that the algorithm shown in Figure 7-2 is a correct solution to the mastermode/normalmode problem?

(4) In the example problems solved in Chapters 4 and 8, each processor had its own copy of the routines which implement the MASTERMODE and NORMALMODE operations. It should be possible, though, for the processors to conserve storage space by sharing these routines, provided appropriate measures are taken to preserve the integrity of each processor's private storage locations. Under what circumstances can such sharing be successfully carried out, and what are the details of the necessary arrangements?

(5) The output data for the example problems mentioned above consisted of a sequence of reports of significant events by the various processors. In a complicated problem it is a long and tedious job to manually inspect the output data, such as that appearing on pages C-8 through C-10, to determine whether the processors behaved correctly. Could we devise an automated checker which would perform this function after being given a formal description of the coordination problem, a description of the processor configuration for a specific experimental run, and a listing of the resultant output data?

9.3 FINAL REMARKS.

Before concluding this dissertation the author would like to express his appreciation to the patient and diligent reader who has followed the entire presentation, especially through the difficult sections of the correctness proof in

Chapter 7. It is hoped that this investigation of low-level multiple processor coordination may provide some inspiration to those readers who are or will be faced with the task of organizing and distributing the computational resources of a large-scale data processing system.

Bibliography

Ak72 Akkoyunlu, A. et al. "An Operating System for a Network Environment," in Proc. Symp. Comput. Commun. Networks and Teletraffic, 22:529-538. Brooklyn, New York: Polytechnic Inst. Brooklyn, April 1972.

Be75 Bernstein, A. J. and P. Siegel. "A Computer Architecture for Level Structured Systems." IEEE Trans. Computers, C-24/8:785-793 (August 1975).

Br70 Brinch Hansen, P. "The Nucleus of a Multiprogramming System." Comm. ACM, 13/4:238-250 (April 1970).

Br72 -----. "A Comparison of Two Synchronizing Concepts." Acta Informatica, 1:190-199 (1972).

Co71 Courtois, P. J., F. Heymans, and D. L. Parnas. "Concurrent Control with Readers and Writers." Comm. ACM, 14/10:667-668 (October 1971).

De67 deBruijn, N. G. "Additional Comments on a Problem in Concurrent Programming Control." Comm. ACM, 10/3:137-138 (March 1967).

De74 DECsystem-10 Monitor Calls Manual, Fourth Revision. Maynard, Mass.: Digital Equipment Corporation, May 1974.

De75 DECsystem-10 System Reference Manual, Third Edition. Maynard, Mass.: Digital Equipment Corporation, 1975.

Di65 Dijkstra, E. W. "Solution of a Problem in Concurrent Programming Control." Comm. ACM, 9/5:321-322 (May 1966).

Di68a -----. "The Structure of the THE Multiprogramming System." Comm. ACM, 11/5:341-346 (May 1968).

Di68b -----. "Cooperating Sequential Processes," in Programming Languages, edited by F. Genuys. New York: Academic Press, 1968.

Di68c -----. "A Constructive Approach to the Problem of Program Correctness." BIT, 8:174-186 (1968).

Di71 -----. "Hierarchical Ordering of Sequential Processes." Acta Informatica, 1:115-138 (1971).

Ei72 Eisenberg, M. A. and M. R. McGuire. "Further Comments on Dijkstra's Concurrent Programming Control Problem." Comm. ACM, 15/11:999 (November 1972).

Ha67 Habermann, A. N. On the Harmonious Cooperation of Abstract Machines. PhD Thesis. Eindhoven, The Netherlands: Technological University of Eindhoven, 1967.

Ha72 -----. "Synchronization of Communicating Processes." Comm. ACM, 15/3:171-176 (March 1972).

Ho74 Hoare, C. A. R. "Monitors: An Operating System Structuring Concept." Comm. ACM, 17/10:549-557 (October 1974).

Kn66 Knuth, D. E. "Additional Comments on a Problem in Concurrent Programming Control." Comm. ACM, 9/5:321-322 (May 1966).

Kn73 -----. The Art of Computer Programming, Volume 1, Second Edition. Reading, Mass.: Addison-Wesley, 1973.

La74 Lamport, L. "A New Solution of Dijkstra's Concurrent Programming Problem." Comm. ACM, 17/8:453-455 (August 1974).

Le72 Levitt, K. N. "The Application of Program-Proving Techniques to the Verification of Synchronization Processes," in Proceedings 1972 FJCC, 41:33-47. Montvale, N.J.: AFIPS Press, 1972.

Li72 Liskov, B. H. "The Design of the Venus Operating System." Comm. ACM, 15/3:144-149 (March 1972).

Li73 Lipton, R. J. On Synchronizing Primitive Systems. PhD Thesis. Pittsburgh: Carnegie-Mellon University, June 1973.

Lo72 Lorin, H. Parallelism in Hardware and Software: Real and Apparent Concurrency. Englewood Cliffs, N. J.: Prentice-Hall, 1972.

Pr75 Presser, L. "Multiprogramming Coordination." Computing Surveys, 7/1:21-44 (March 1975).

Sa66 Saltzer, J. H. Traffic Control in a Multiplexed Computer System. Project MAC Report MAC-TR-30. Cambridge, Mass: Massachusetts Inst. Technology, July 1966.

Se72 Sevcik, K. et al. "Project SUE as a Learning Experience," in Proceedings 1972 FJCC, 41:331-339. Montvale, N. J.: AFIPS Press, 1972.

Va72 Vantilborgh, H. and A. vanLamsweerde. "On an Extension of Dijkstra's Semaphore Primitives." Information Processing Letters, 1:181-186 (1972).

APPENDIX A

THE BAKERY ALGORITHM

The following solution to the critical section problem was given by Leslie Lamport in the Communications of the ACM, August 1974. The algorithm is based upon one commonly used in bakeries, in which a customer receives a number upon entering the store. The holder of the lowest number is the next one served. In Lamport's algorithm each processor chooses its own number. The processors are named 1, 2, . . ., N. If two processors choose the same number, then the one with the lowest name goes first. The memory locations shared by the processors consist of the integer arrays CHOOSING[1:N] and NUMBER[1:N]. Each processor has a private integer variable J which is used as a loop index. The following is the program for Processor I:

```
BEGIN
    L1: CHOOSING[I]:=1;
        NUMBER[I]:=1+MAXIMUM(NUMBER[1], . . ., NUMBER[N]);
        CHOOSING[I]:=0;
        FOR J=1 UNTIL N DO
            BEGIN
              L2: IF CHOOSING[J]≠0 THEN GOTO L2;
              L3: IF NUMBER[J]≠0 AND
                     (NUMBER[J]<NUMBER[I] OR
                     (NUMBER[J]=NUMBER[I] AND J<I))
                  THEN GOTO L3;
            END;
        CRITICAL SECTION;
        NUMBER[I]:=0;
        NONCRITICAL SECTION;
        GOTO L1;
END
```

APPENDIX B

IMPLEMENTATION OF MASTERMODE/NORMALMODE ALGORITHM

```
;       MACRO-10 ASSEMBLY LANGUAGE ROUTINES INVOKED
;       BY CALLS TO THE MASTERMODE AND NORMALMODE
;       PROCEDURES.  LOCAL CONSTANTS WAKEUP, FLAG,
;       AND JOBNO WILL HAVE ALREADY BEEN INITIALIZED
;       BY A CALL TO THE INITIALIZE PROCEDURE.

MAST:   SETZM   @WAKEUP             ;CLEAR OUR WAKEUP WORD.
        MOVE    AC,WAKEUP           ;LOAD BOTH HALVES OF AC WITH
        HRLS    AC                  ;ADDRESS OF OUR WAKEUP WORD
        EXCH    AC,@FLAG            ;AND SWAP WITH FLAG WORD.
        TRNN    AC,-1               ;DO WE HAVE A LEADER?
        POPJ    P,                  ;NO, RETURN (IN MASTERMODE).
        HLRZ    J,AC                ;YES, SO LOOK AT HIS WAKEUP
        MOVE    AC,JOBNO            ;WORD TO SEE IF HE HAS
        EXCH    AC,(J)              ;GOTTEN TOO FAR AHEAD
        JUMPE   AC,MAST1            ;AND NEEDS A WAKEUP.
        WAKE    AC,                 ;HE HAS, SO WAKE HIM UP.
          HALT                      ;(ERROR)
MAST1:  MOVSI   AC,(1B16!1B17)      ;WE SLEEP HERE UNTIL
        HIBER   AC,                 ;AWAKENED BY OUR LEADER.
          HALT                      ;(ERROR)
        POPJ    P,                  ;RETURN (IN MASTERMODE).

NORM:   HRRZ    AC,WAKEUP           ;LOAD RIGHT HALF OF AC AND
        XORB    AC,@FLAG            ;EXCLUSIVE OR WITH FLAG WORD
        TRNN    AC,-1               ;DO WE HAVE A FOLLOWER?
        POPJ    P,                  ;NO, RETURN (IN NORMALMODE).
        HLRS    @FLAG               ;YES, RESTORE FLAG WORD.
        MOVE    AC,JOBNO            ;LOOK AT OUR OWN WAKEUP WORD
        EXCH    AC,@WAKEUP          ;TO SEE IF WE'VE GOTTEN TOO
        JUMPN   AC,NORM1            ;FAR AHEAD OF OUR FOLLOWER.
        MOVSI   AC,(1B16!1B17)      ;WE HAVE, SO WE WILL SLEEP
        HIBER   AC,                 ;HERE UNTIL HE CATCHES UP.
          HALT                      ;(ERROR)
        MOVE    AC,@WAKEUP          ;NOW HIS NUMBER IS THERE.
NORM1:  WAKE    AC,                 ;WAKE OUR FOLLOWER UP.
          HALT                      ;(ERROR)
        POPJ    P,                  ;RETURN (IN NORMALMODE).
```

APPENDIX C

LISTINGS FOR THE PRODUCER/CONSUMER/DISTRIBUTOR PROBLEM

```
BEGIN   ! PRODUCER FOR BUFFER POOL PROBLEM;

  INTEGER  NPROD,NCON,NBUF,P.LOOP;
  STRING   GROUP;
  READ(GROUP,NPROD,NCON,NBUF,P.LOOP);

BEGIN

  INTEGER  MYNUM,MYBUF,BASE,HEAD,USERBASE,J;
  BOOLEAN  FOUND,MUST.WAIT;
  INTEGER ARRAY  P.USE[1:NPROD], C.USE[1:NCON],
                 P.LINE[-1:NPROD], C.LINE[-1:NCON],
                 FULL[1:NBUF], BUSY[1:NBUF];
  EXTERNAL PROCEDURE  MASTERMODE, NORMALMODE, INITIALIZE,
                      ASSIGN, SET, PUT.IN, REMOVE.FROM,
                      REPORT, PAUSE, SLEEP, WAKEUP;
  EXTERNAL INTEGER PROCEDURE  VALUE.OF, RANDOM;
  EXTERNAL BOOLEAN PROCEDURE  TRULY;

  ! ESTABLISH BASE ADDRESS OF SHARED AREA;
  IF GROUP="A" THEN BASE:=0;
  IF GROUP="B" THEN BASE:=100;
  IF GROUP="C" THEN BASE:=200;
  IF GROUP="D" THEN BASE:=300;
  USERBASE:=2*(NPROD+NCON)+BASE;

  ! ESTABLISH POINTERS TO SHARED VARIABLES;
  FOR J:= 1 UNTIL NBUF  DO FULL[J]:=USERBASE+J;
  FOR J:= 1 UNTIL NBUF  DO BUSY[J]:=FULL[NBUF]+J;
  FOR J:= 1 UNTIL NPROD DO P.USE[J]:=BUSY[NBUF]+J;
  FOR J:= 1 UNTIL NCON  DO C.USE[J]:=P.USE[NPROD]+J;
  FOR J:=-1 UNTIL NPROD DO P.LINE[J]:=C.USE[NCON]+J+2;
  FOR J:=-1 UNTIL NCON  DO C.LINE[J]:=P.LINE[NPROD]+J+2;

  READ(MYNUM);  INITIALIZE(MYNUM,BASE);
  FOR J:=1 UNTIL P.LOOP DO
```

```
BEGIN ! START OF PRODUCER CYCLE;

      PAUSE(500+(5*RANDOM));

  MASTERMODE;
      MYBUF:=0;
      FOUND:=FALSE;
      WHILE MYBUF<NBUF AND NOT FOUND DO
          BEGIN ! SEARCH FOR A BUFFER;
                MYBUF:=MYBUF+1;
                IF NOT TRULY(FULL[MYBUF]) AND
                   NOT TRULY(BUSY[MYBUF]) THEN
                   BEGIN FOUND:=TRUE;
                         SET(BUSY[MYBUF],TRUE);
                         ASSIGN(P.USE[MYNUM],MYBUF);
                         MUST.WAIT:=FALSE;
                   END;
          END OF SEARCH LOOP;
      IF NOT FOUND THEN
          BEGIN REPORT(GROUP,"PRODUCER",MYNUM,
                       "WAITING TO FILL  ",0);
                PUT.IN(P.LINE,MYNUM);
                MUST.WAIT:=TRUE;
          END;
  NORMALMODE;

      IF MUST.WAIT THEN SLEEP;
      MYBUF:=VALUE.OF(P.USE[MYNUM]);
      REPORT(GROUP,"PRODUCER",MYNUM,
             "STARTING TO FILL ",MYBUF);
      PAUSE(500+RANDOM);

  MASTERMODE;
      REPORT(GROUP,"PRODUCER",MYNUM,
             "FINISHED FILLING ",MYBUF);
      HEAD:=VALUE.OF(C.LINE[0]);
      IF HEAD>0 THEN BEGIN ASSIGN(C.USE[HEAD],MYBUF);
                           REMOVE.FROM(C.LINE);
                           WAKEUP(HEAD+NPROD);
                     END
                ELSE BEGIN SET(FULL[MYBUF],TRUE);
                           SET(BUSY[MYBUF],FALSE);
                           END;
  NORMALMODE;

 END OF CYCLE;

END;
END
```

```
BEGIN   ! CONSUMER FOR BUFFER POOL PROBLEM;

  INTEGER  NPROD,NCON,NBUF,C.LOOP;
  STRING   GROUP;
  READ(GROUP,NPROD,NCON,NBUF,C.LOOP);

BEGIN

  INTEGER  MYNUM,MYBUF,BASE,HEAD,USERBASE,J;
  BOOLEAN  FOUND,MUST.WAIT;
  INTEGER ARRAY  P.USE[1:NPROD], C.USE[1:NCON],
                 P.LINE[-1:NPROD], C.LINE[-1:NCON],
                 FULL[1:NBUF], BUSY[1:NBUF];
  EXTERNAL PROCEDURE  MASTERMODE, NORMALMODE, INITIALIZE,
                      ASSIGN, SET, PUT.IN, REMOVE.FROM,
                      REPORT, PAUSE, SLEEP, WAKEUP;
  EXTERNAL INTEGER PROCEDURE  VALUE.OF, RANDOM;
  EXTERNAL BOOLEAN PROCEDURE  TRULY;

  ! ESTABLISH BASE ADDRESS OF SHARED AREA;
  IF GROUP="A" THEN BASE:=0;
  IF GROUP="B" THEN BASE:=100;
  IF GROUP="C" THEN BASE:=200;
  IF GROUP="D" THEN BASE:=300;
  USERBASE:=2*(NPROD+NCON)+BASE;

  ! ESTABLISH POINTERS TO SHARED VARIABLES;
  FOR J:= 1 UNTIL NBUF  DO FULL[J]:=USERBASE+J;
  FOR J:= 1 UNTIL NBUF  DO BUSY[J]:=FULL[NBUF]+J;
  FOR J:= 1 UNTIL NPROD DO P.USE[J]:=BUSY[NBUF]+J;
  FOR J:= 1 UNTIL NCON  DO C.USE[J]:=P.USE[NPROD]+J;
  FOR J:=-1 UNTIL NPROD DO P.LINE[J]:=C.USE[NCON]+J+2;
  FOR J:=-1 UNTIL NCON  DO C.LINE[J]:=P.LINE[NPROD]+J+2;

  READ(MYNUM);  INITIALIZE(MYNUM+NPROD,BASE);
  FOR J:=1 UNTIL C.LOOP DO
```

```
BEGIN ! START OF CONSUMER CYCLE;

      PAUSE(500+(5*RANDOM));

  MASTERMODE;
      MYBUF:=0;
      FOUND:=FALSE;
      WHILE MYBUF<NBUF AND NOT FOUND DO
          BEGIN ! SEARCH FOR A BUFFER;
                MYBUF:=MYBUF+1;
                IF TRULY(FULL[MYBUF]) THEN
                   BEGIN FOUND:=TRUE;
                         SET(FULL[MYBUF],FALSE);
                         SET(BUSY[MYBUF],TRUE);
                         ASSIGN(C.USE[MYNUM],MYBUF);
                         MUST.WAIT:=FALSE;
                   END;
          END OF SEARCH LOOP;
      IF NOT FOUND THEN
          BEGIN REPORT(GROUP,"CONSUMER",MYNUM,
                       "WAITING TO EMPTY ",0);
                PUT.IN(C.LINE,MYNUM);
                MUST.WAIT:=TRUE;
          END;
  NORMALMODE;

      IF MUST.WAIT THEN SLEEP;
      MYBUF:=VALUE.OF(C.USE[MYNUM]);
      REPORT(GROUP,"CONSUMER",MYNUM,
             "STARTING TO EMPTY",MYBUF);
      PAUSE(500+RANDOM);

  MASTERMODE;
      REPORT(GROUP,"CONSUMER",MYNUM,
             "FINISHED EMPTYING",MYBUF);
      HEAD:=VALUE.OF(P.LINE[0]);
      IF HEAD>0 THEN BEGIN ASSIGN(P.USE[HEAD],MYBUF);
                           REMOVE.FROM(P.LINE);
                           WAKEUP(HEAD);
                     END
                ELSE SET(BUSY[MYBUF],FALSE);
  NORMALMODE;

 END OF CYCLE;

END;
END
```

```
BEGIN   ! DISTRIBUTOR FOR BUFFER POOL PROBLEM;

  INTEGER  NPRODA,NCONA,NBUFA,NPRODB,NCONB,NBUFB,
           NPRODC,NCONC,NBUFC,NDIS,D.LOOP;
  READ(NPRODA,NCONA,NBUFA);
  READ(NPRODB,NCONB,NBUFB);
  READ(NPRODC,NCONC,NBUFC);
  READ(NDIS,D.LOOP);

BEGIN

  INTEGER  MYNAME,MYNUM,MYBUF,BASE,HEAD,USERBASE,
           NPROD,NBOTH,NCON,NBUF,J,JX;
  BOOLEAN  FOUND,MUST.WAIT;
  STRING   GROUP;
  INTEGER ARRAY  P.LINE[-1:NDIS+IMAX(NPRODA,NPRODB,NPRODC)],
                 C.LINE[-1:IMAX(NCONA,NCONB,NCONC)],
                 P.USE[1:NDIS+IMAX(NPRODA,NPRODB,NPRODC)],
                 C.USE[1:IMAX(NCONA,NCONB,NCONC)],
                 FULL[1:IMAX(NBUFA,NBUFB,NBUFC)],
                 BUSY[1:IMAX(NBUFA,NBUFB,NBUFC)];
  EXTERNAL PROCEDURE  MASTERMODE, NORMALMODE, INITIALIZE,
                      ASSIGN, SET, PUT.IN, REMOVE.FROM,
                      REPORT, PAUSE, SLEEP, WAKEUP, NEWGROUP;
  EXTERNAL INTEGER PROCEDURE  VALUE.OF, RANDOM;
  EXTERNAL BOOLEAN PROCEDURE  TRULY;

  ! PROCEDURE TO ESTABLISH POINTERS TO SHARED VARIABLES;
  PROCEDURE SET.POINTERS(S); VALUE S; STRING S;
  BEGIN GROUP:=S;
        IF S="A" THEN BEGIN BASE:=000; NPROD:=NPRODA;
                            NCON:=NCONA; NBUF:=NBUFA; END;
        IF S="B" THEN BEGIN BASE:=100; NPROD:=NPRODB;
                            NCON:=NCONB; NBUF:=NBUFB; END;
        IF S="C" THEN BEGIN BASE:=200; NPROD:=NPRODC;
                            NCON:=NCONC; NBUF:=NBUFC; END;
        NBOTH:=NDIS+NPROD;
        MYNUM:=MYNAME+NPROD;
        USERBASE:=2*(NBOTH+NCON)+BASE;
        FOR J:= 1 UNTIL NBUF  DO FULL[J]:=USERBASE+J;
        FOR J:= 1 UNTIL NBUF  DO BUSY[J]:=FULL[NBUF]+J;
        FOR J:= 1 UNTIL NBOTH DO P.USE[J]:=BUSY[NBUF]+J;
        FOR J:= 1 UNTIL NCON  DO C.USE[J]:=P.USE[NBOTH]+J;
        FOR J:=-1 UNTIL NBOTH DO P.LINE[J]:=C.USE[NCON]+J+2;
        FOR J:=-1 UNTIL NCON  DO C.LINE[J]:=P.LINE[NBOTH]+J+2;
  END;


  READ(MYNAME);  SET.POINTERS("A");
  INITIALIZE(MYNUM,BASE);
  FOR JX:=1 UNTIL D.LOOP DO
```

```
BEGIN ! START OF DISTRIBUTOR CYCLE;

      PAUSE(500+(5*RANDOM));

  MASTERMODE;
      MYBUF:=0;
      FOUND:=FALSE;
      WHILE MYBUF<NBUF AND NOT FOUND DO
          BEGIN ! SEARCH FOR A BUFFER;
                MYBUF:=MYBUF+1;
                IF NOT TRULY(FULL[MYBUF]) AND
                   NOT TRULY(BUSY[MYBUF]) THEN
                   BEGIN FOUND:=TRUE;
                         SET(BUSY[MYBUF],TRUE);
                         ASSIGN(P.USE[MYNUM],MYBUF);
                         MUST.WAIT:=FALSE;
                   END;
          END OF SEARCH LOOP;
      IF NOT FOUND THEN
          BEGIN REPORT(GROUP,"DSTRBUTR",MYNAME,
                       "WAITING TO FILL  ",0);
                PUT.IN(P.LINE,MYNUM);
                MUST.WAIT:=TRUE;
          END;
  NORMALMODE;

      IF MUST.WAIT THEN SLEEP;
      MYBUF:=VALUE.OF(P.USE[MYNUM]);
      REPORT(GROUP,"DSTRBUTR",MYNAME,
             "STARTING TO FILL ",MYBUF);
      PAUSE(500+RANDOM);

  MASTERMODE;
      REPORT(GROUP,"DSTRBUTR",MYNAME,
             "FINISHED FILLING ",MYBUF);
      HEAD:=VALUE.OF(C.LINE[0]);
      IF HEAD>0 THEN BEGIN ASSIGN(C.USE[HEAD],MYBUF);
                           REMOVE.FROM(C.LINE);
                           WAKEUP(HEAD+NBOTH);
                     END
                ELSE BEGIN SET(FULL[MYBUF],TRUE);
                           SET(BUSY[MYBUF],FALSE);
                     END;
  NORMALMODE;

      IF GROUP="A" THEN SET.POINTERS("B") ELSE
      IF GROUP="B" THEN SET.POINTERS("C") ELSE
      IF GROUP="C" THEN SET.POINTERS("A");
      NEWGROUP(MYNUM,BASE);

  END OF CYCLE;

END;
END
```

```
GROUP B CONSUMER 1 WAITING TO EMPTY  BUFFER   AT 3766.225
GROUP A PRODUCER 2 STARTING TO FILL  BUFFER 1 AT 3766.245
GROUP A PRODUCER 1 STARTING TO FILL  BUFFER 2 AT 3766.272
GROUP A PRODUCER 2 FINISHED FILLING  BUFFER 1 AT 3766.280
GROUP C CONSUMER 1 WAITING TO EMPTY  BUFFER   AT 3766.303
GROUP A PRODUCER 1 FINISHED FILLING  BUFFER 2 AT 3766.338
GROUP C CONSUMER 2 WAITING TO EMPTY  BUFFER   AT 3766.349
GROUP A DSTRBUTR 2 STARTING TO FILL  BUFFER 3 AT 3766.362
GROUP A DSTRBUTR 1 WAITING TO FILL   BUFFER   AT 3766.366
GROUP A CONSUMER 1 STARTING TO EMPTY BUFFER 1 AT 3766.386
GROUP A PRODUCER 2 WAITING TO FILL   BUFFER   AT 3766.388
GROUP A DSTRBUTR 2 FINISHED FILLING  BUFFER 3 AT 3766.405
GROUP B CONSUMER 2 WAITING TO EMPTY  BUFFER   AT 3766.407
GROUP B CONSUMER 3 WAITING TO EMPTY  BUFFER   AT 3766.414
GROUP B PRODUCER 1 STARTING TO FILL  BUFFER 1 AT 3766.418
GROUP A PRODUCER 1 WAITING TO FILL   BUFFER   AT 3766.429
GROUP A CONSUMER 1 FINISHED EMPTYING BUFFER 1 AT 3766.454
GROUP A DSTRBUTR 1 STARTING TO FILL  BUFFER 1 AT 3766.459
GROUP B PRODUCER 1 FINISHED FILLING  BUFFER 1 AT 3766.468
GROUP B CONSUMER 1 STARTING TO EMPTY BUFFER 1 AT 3766.471
GROUP A DSTRBUTR 1 FINISHED FILLING  BUFFER 1 AT 3766.508
GROUP A CONSUMER 1 STARTING TO EMPTY BUFFER 1 AT 3766.519
GROUP B PRODUCER 1 STARTING TO FILL  BUFFER 2 AT 3766.554
GROUP B CONSUMER 1 FINISHED EMPTYING BUFFER 1 AT 3766.556
GROUP B DSTRBUTR 2 STARTING TO FILL  BUFFER 1 AT 3766.560
GROUP A CONSUMER 1 FINISHED EMPTYING BUFFER 1 AT 3766.569
GROUP B PRODUCER 1 FINISHED FILLING  BUFFER 2 AT 3766.589
GROUP B CONSUMER 2 STARTING TO EMPTY BUFFER 2 AT 3767.113
GROUP B DSTRBUTR 2 FINISHED FILLING  BUFFER 1 AT 3767.113
GROUP B CONSUMER 3 STARTING TO EMPTY BUFFER 1 AT 3767.117
GROUP B DSTRBUTR 1 STARTING TO FILL  BUFFER 3 AT 3767.118
GROUP B CONSUMER 1 WAITING TO EMPTY  BUFFER   AT 3767.121
GROUP B DSTRBUTR 1 FINISHED FILLING  BUFFER 3 AT 3767.158
GROUP B CONSUMER 1 STARTING TO EMPTY BUFFER 3 AT 3767.161
GROUP B PRODUCER 1 STARTING TO FILL  BUFFER 4 AT 3767.174
GROUP B CONSUMER 3 FINISHED EMPTYING BUFFER 1 AT 3767.185
GROUP C DSTRBUTR 1 STARTING TO FILL  BUFFER 1 AT 3767.194
GROUP B CONSUMER 2 FINISHED EMPTYING BUFFER 2 AT 3767.204
GROUP B PRODUCER 1 FINISHED FILLING  BUFFER 4 AT 3767.235
GROUP C DSTRBUTR 2 WAITING TO FILL   BUFFER   AT 3767.240
GROUP C DSTRBUTR 1 FINISHED FILLING  BUFFER 1 AT 3767.270
GROUP B CONSUMER 2 STARTING TO EMPTY BUFFER 4 AT 3767.275
GROUP B PRODUCER 1 STARTING TO FILL  BUFFER 1 AT 3767.311
GROUP B PRODUCER 1 FINISHED FILLING  BUFFER 1 AT 3767.397
GROUP B PRODUCER 1 STARTING TO FILL  BUFFER 2 AT 3767.473
GROUP B CONSUMER 3 STARTING TO EMPTY BUFFER 1 AT 3767.479
GROUP B PRODUCER 1 FINISHED FILLING  BUFFER 2 AT 3767.526
GROUP A PRODUCER 2 STARTING TO FILL  BUFFER 1 AT 3767.537
GROUP A PRODUCER 2 FINISHED FILLING  BUFFER 1 AT 3767.586
GROUP A PRODUCER 2 WAITING TO FILL   BUFFER   AT 3767.642
GROUP A CONSUMER 1 STARTING TO EMPTY BUFFER 1 AT 3767.799
GROUP B CONSUMER 1 FINISHED EMPTYING BUFFER 3 AT 3767.812
GROUP A CONSUMER 1 FINISHED EMPTYING BUFFER 1 AT 3767.884
GROUP A PRODUCER 1 STARTING TO FILL  BUFFER 1 AT 3767.886
GROUP A PRODUCER 1 FINISHED FILLING  BUFFER 1 AT 3767.956
```

```
GROUP B CONSUMER 1 STARTING TO EMPTY BUFFER 2 AT 3767.963
GROUP B CONSUMER 2 FINISHED EMPTYING BUFFER 4 AT 3767.979
GROUP B CONSUMER 1 FINISHED EMPTYING BUFFER 2 AT 3768.015
GROUP A CONSUMER 1 STARTING TO EMPTY BUFFER 1 AT 3768.018
GROUP A CONSUMER 1 FINISHED EMPTYING BUFFER 1 AT 3768.093
GROUP B CONSUMER 3 FINISHED EMPTYING BUFFER 1 AT 3768.194
GROUP B CONSUMER 2 WAITING TO EMPTY  BUFFER   AT 3768.229
GROUP C CONSUMER 1 STARTING TO EMPTY BUFFER 1 AT 3768.356
GROUP C CONSUMER 1 FINISHED EMPTYING BUFFER 1 AT 3768.432
GROUP C DSTRBUTR 2 STARTING TO FILL  BUFFER 1 AT 3768.435
GROUP B CONSUMER 3 WAITING TO EMPTY  BUFFER   AT 3768.470
GROUP C CONSUMER 1 WAITING TO EMPTY  BUFFER   AT 3768.472
GROUP C DSTRBUTR 2 FINISHED FILLING  BUFFER 1 AT 3768.500
GROUP C CONSUMER 2 STARTING TO EMPTY BUFFER 1 AT 3768.564
GROUP C CONSUMER 2 FINISHED EMPTYING BUFFER 1 AT 3768.637
GROUP C CONSUMER 2 WAITING TO EMPTY  BUFFER   AT 3768.774
GROUP A PRODUCER 2 STARTING TO FILL  BUFFER 1 AT 3768.993
GROUP A DSTRBUTR 1 WAITING TO FILL   BUFFER   AT 3768.995
GROUP A DSTRBUTR 2 WAITING TO FILL   BUFFER   AT 3768.998
GROUP A PRODUCER 2 FINISHED FILLING  BUFFER 1 AT 3769.053
GROUP A CONSUMER 1 STARTING TO EMPTY BUFFER 1 AT 3769.067
GROUP A CONSUMER 1 FINISHED EMPTYING BUFFER 1 AT 3769.119
GROUP A DSTRBUTR 1 STARTING TO FILL  BUFFER 1 AT 3769.121
GROUP A DSTRBUTR 1 FINISHED FILLING  BUFFER 1 AT 3769.171
GROUP A CONSUMER 1 STARTING TO EMPTY BUFFER 1 AT 3769.260
GROUP A CONSUMER 1 FINISHED EMPTYING BUFFER 1 AT 3769.322
GROUP A DSTRBUTR 2 STARTING TO FILL  BUFFER 1 AT 3769.324
GROUP A CONSUMER 1 STARTING TO EMPTY BUFFER 2 AT 3769.379
GROUP A DSTRBUTR 2 FINISHED FILLING  BUFFER 1 AT 3769.390
GROUP B DSTRBUTR 1 STARTING TO FILL  BUFFER 1 AT 3769.406
GROUP A CONSUMER 1 FINISHED EMPTYING BUFFER 2 AT 3769.423
GROUP B DSTRBUTR 2 STARTING TO FILL  BUFFER 2 AT 3769.639
GROUP B DSTRBUTR 1 FINISHED FILLING  BUFFER 1 AT 3769.809
GROUP B CONSUMER 2 STARTING TO EMPTY BUFFER 1 AT 3769.815
GROUP B CONSUMER 2 FINISHED EMPTYING BUFFER 1 AT 3769.859
GROUP C DSTRBUTR 1 STARTING TO FILL  BUFFER 1 AT 3769.968
GROUP C DSTRBUTR 1 FINISHED FILLING  BUFFER 1 AT 3770.003
GROUP C CONSUMER 1 STARTING TO EMPTY BUFFER 1 AT 3770.005
GROUP C CONSUMER 1 FINISHED EMPTYING BUFFER 1 AT 3770.049
GROUP B DSTRBUTR 2 FINISHED FILLING  BUFFER 2 AT 3770.149
GROUP B CONSUMER 3 STARTING TO EMPTY BUFFER 2 AT 3770.153
GROUP B CONSUMER 3 FINISHED EMPTYING BUFFER 2 AT 3770.234
GROUP A DSTRBUTR 1 STARTING TO FILL  BUFFER 2 AT 3770.316
GROUP B CONSUMER 3 WAITING TO EMPTY  BUFFER   AT 3770.340
GROUP A DSTRBUTR 1 FINISHED FILLING  BUFFER 2 AT 3770.350
GROUP C DSTRBUTR 2 STARTING TO FILL  BUFFER 1 AT 3770.364
GROUP C DSTRBUTR 2 FINISHED FILLING  BUFFER 1 AT 3770.429
GROUP C CONSUMER 2 STARTING TO EMPTY BUFFER 1 AT 3770.432
GROUP C CONSUMER 2 FINISHED EMPTYING BUFFER 1 AT 3770.498
GROUP A DSTRBUTR 2 WAITING TO FILL   BUFFER   AT 3770.539
GROUP A CONSUMER 1 STARTING TO EMPTY BUFFER 1 AT 3770.606
GROUP C CONSUMER 2 WAITING TO EMPTY  BUFFER   AT 3770.633
GROUP A CONSUMER 1 FINISHED EMPTYING BUFFER 1 AT 3770.647
GROUP A DSTRBUTR 2 STARTING TO FILL  BUFFER 1 AT 3770.650
GROUP A CONSUMER 1 STARTING TO EMPTY BUFFER 2 AT 3770.682
```

```
GROUP A CONSUMER 1 FINISHED EMPTYING BUFFER 2 AT 3770.771
GROUP A DSTRBUTR 2 FINISHED FILLING  BUFFER 1 AT 3771.187
GROUP A CONSUMER 1 STARTING TO EMPTY BUFFER 1 AT 3771.253
GROUP A CONSUMER 1 FINISHED EMPTYING BUFFER 1 AT 3771.288
GROUP B DSTRBUTR 2 STARTING TO FILL  BUFFER 1 AT 3771.365
GROUP B DSTRBUTR 2 FINISHED FILLING  BUFFER 1 AT 3771.414
GROUP B CONSUMER 3 STARTING TO EMPTY BUFFER 1 AT 3771.420
GROUP A CONSUMER 1 STARTING TO EMPTY BUFFER 3 AT 3771.434
GROUP B CONSUMER 3 FINISHED EMPTYING BUFFER 1 AT 3771.456
GROUP A CONSUMER 1 FINISHED EMPTYING BUFFER 3 AT 3771.510
GROUP C DSTRBUTR 2 STARTING TO FILL  BUFFER 1 AT 3771.536
GROUP C DSTRBUTR 2 FINISHED FILLING  BUFFER 1 AT 3771.602
GROUP C CONSUMER 2 STARTING TO EMPTY BUFFER 1 AT 3771.605
GROUP C CONSUMER 2 FINISHED EMPTYING BUFFER 1 AT 3771.657
```

VITA

Bob Edward Baker was born on 18 February 1944 in McAlester, Oklahoma. After graduating from Stillwater High School, Stillwater, Oklahoma in 1961, he attended Eastern Oklahoma State College for one year. He entered Oklahoma State University in 1962 and was graduated with the Bachelor of Science degree in Electrical Engineering. Upon graduation he was granted a regular commission in the U. S. Air Force and was assigned to the Air Force Institute of Technology (AFIT) where he obtained a Master's degree in the Graduate Guidance and Control program. Between 1968 and 1972 he was assigned as an electronics engineer to the Nuclear Instrumentation Section, McClellan Central Laboratory, McClellan AFB, California. In 1972 he returned to AFIT as a candidate for the Doctor of Philosophy degree. After completing his PhD coursework in 1974 he was assigned to the System Avionics Division, Air Force Avionics Laboratory, where his present position is Chief, Advanced Systems Group, System Simulation Branch.

Permanent address: Route 2, Box 164A
Wilburton, Oklahoma 74578

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: DS/EE/76-1	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER: Doctoral Thesis	
4. TITLE (and Subtitle): EFFICIENT MULTIPLE PROCESSOR COORDINATION	5. TYPE OF REPORT & PERIOD COVERED: PhD Dissertation
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Bob E. Baker, Captain, USAF	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Air Force Institute of Technology (AFIT-EN), Wright-Patterson AFB, Ohio 45433	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: PE 62204F, Project 20030319
11. CONTROLLING OFFICE NAME AND ADDRESS: AFAL/AAF, Air Force Avionics Laboratory, Wright-Patterson AFB, Ohio 45433	12. REPORT DATE: Nov 76
	13. NUMBER OF PAGES: 236
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): 237 p.	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

16 2003 17 03

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Approved for public release; IAW AFR 190-17

JERRAL F. GUESS, CAPT, USAF
Director of Information

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Multiprocessing
Multiprogramming
Critical section problem
Process coordination
Process synchronization

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In recent years the concept of level-structured system organization has received growing acceptance as a computer operating system design technique. This approach treats the hardware of a computing system as the bottom level of a multilevel system which will be built up one level at a time. Each level creates an abstraction of the next lower level by implementing a new "virtual machine" which provides some feature not previously available.
(continued on back)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

over

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

012225

Block 20 (Abstract) continued.

This dissertation presents efficient solutions to certain problems of processor interaction which must be faced by the system designer at the lower levels of a strictly level-structured operating system. A loop-free algorithm is developed which solves Dijkstra's "critical section" problem, based on a rudimentary processor delay mechanism and the instruction set of a typical large computer of conventional architecture (DECsystem-10). A completely rigorous proof of the algorithm's correctness is given. The algorithm is used as the basis of a systematic procedure for obtaining efficient solutions to general processor coordination problems. The procedure is illustrated by solving several of the well-known reader/writer problems and a more complex new problem, the producer/consumer/distributor problem, which involves simultaneous use of several independent shared resources.